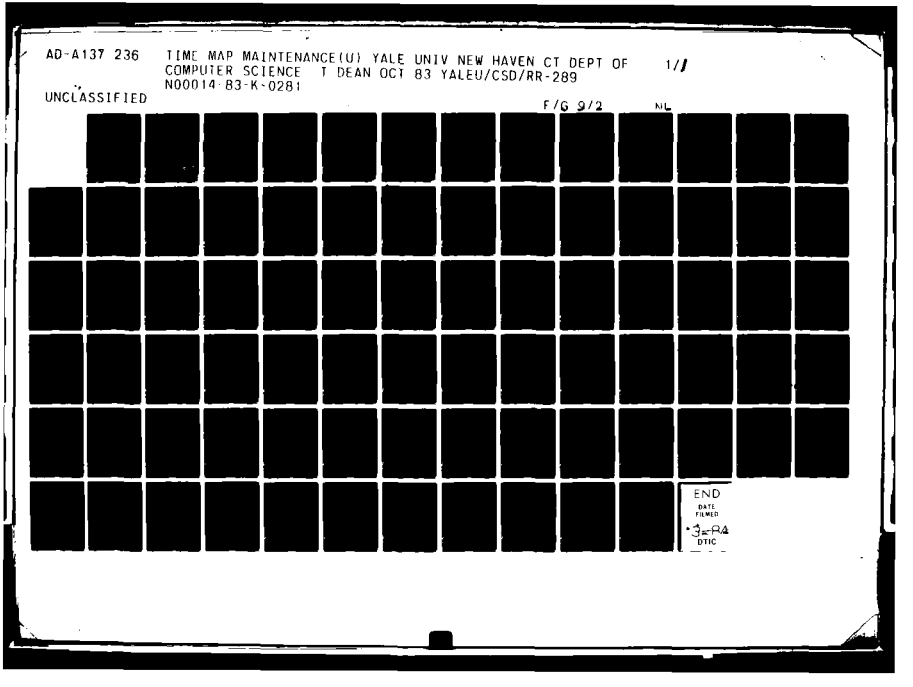
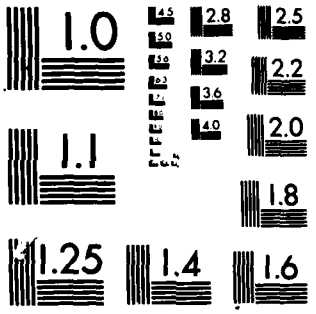

AD-A137 236 TIME MAP MAINTENANCE(U) YALE UNIV NEW HAVEN CT DEPT OF 1/1
COMPUTER SCIENCE T DEAN OCT 83 YALEU/CSD/RR-289
N00014-83-K-0281
UNCLASSIFIED
F/G 9/2 NL
END
DATE FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Time Map Maintenance

Thomas Dean

YaleU/CSD/RR#289

October 1983

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Time Map Maintenance

Thomas Dean

Research Report #289

October 1983

Acknowledgements

Many of the ideas presented in this paper came up in discussions with Drew McDermott, Dave Miller, Yoav Shoham, Stan Letovsky and Eric Gold of the Yale spatial reasoning research group. The term "time map" is due to Drew McDermott.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored under the Office of Naval Research under contract N00014-83-K-0281.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: #289	2. GOVT ACCESSION NO.: AD-A137236
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): Time Map Maintenance	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Thomas Dean	8. CONTRACT OR GRANT NUMBER(s): N00014-83-K-0281
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Yale University, Computer Science Department, 10 Hillhouse Avenue, New Haven, CT 06520	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, VA 22209	12. REPORT DATE: October 1983
	13. NUMBER OF PAGES: 70
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems Program, Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

time map
temporal logic
data dependency
planning

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes a mechanism for dealing with the representation of events and their effects occurring in and over time. The mechanism, which I refer to as a time map manager (TMM), is shown to be useful in problem solvers requiring an ability to reason about time and causality. In addition to describing the theory and its implementation I will demonstrate a programming technique and related discipline based upon the use of data dependencies (Doyle 79a). This technique supports the design of complex control structures capable of recording the conditions under which information is stored and subsequently responding in highly directed ways when those conditions are changed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

-- OFFICIAL DISTIRUBTION LIST --

Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, Virginia 22217	2 copies
Dr. Judith Daly Advanced Research Projects Agency Cybernetics Technology Office 1400 Wilson Boulevard Arlington, Virginia 22209	3 copies
Office of Naval Research Branch Office - Boston 495 Summer Street Boston, Massachusetts 02210	1 copy
Office of Naval Research Branch Office - Chicago 536 South Clark Street Chicago, Illinois 60615	1 copy
Office of Naval Research Branch Office - Pasadena 1030 East Green Street Pasadena, California 91106	1 copy
Mr. Steven Wong New York Area Office 715 Broadway - 5th Floor New York, New York 10003	1 copy
Naval Research Laboratory Technical Information Division Code 2627 Washington, D.C. 20375	6 copies
Dr. A.L. Slafkosky Commandant of the Marine Corps Code RD-1 Washington, D.C. 20380	1 copy
Office of Naval Research Code 455 Arlington, Virginia 22217	1 copy
Office of Naval Research Code 458 Arlington, Virginia 22217	1 copy

Naval Electronics Laboratory Center — 1 copy
Advanced Software Technology Division
Code 5200
San Diego, California 92152

Mr. E.H. Gleissner — 1 copy
Naval Ship Research and Development
Computation and Mathematics Department
Bethesda, Maryland 20084

Captain Grace M. Hopper, USNR — 1 copy
Naval Data Automation Command, Code 00H
Washington Navy Yard
Washington, D.C. 20374

Dr. Robert Engelmore — 2 copies
Advanced Research Project Agency
Information Processing Techniques
1400 Wilson Boulevard
Arlington, Virginia 22209

Professor Omar Wing — 1 copy
Columbia University in the City of New York
Department of Electrical Engineering and
Computer Science
New York, New York 10027

Office of Naval Research — 1 copy
Assistant Chief for Technology
Code 200
Arlington, Virginia 22217

Captain Richard L. Martin, USN — 1 copy
Commanding Officer
USS Francis Marion (LPA-249)
FPO New York 09501

Major J.P. Pennell — 1 copy
Headquarters, Marine Corp.
(Attn: Code CCA-40)
Washington, D.C. 20380

Computer Systems Management, Inc. — 5 copies
1300 Wilson Boulevard, Suite 102
Arlington, Virginia 22209

Ms. Robin Dillard — 1 copy
Naval Ocean Systems Center
C2 Information Processing Branch (Code 8242)
271 Catalina Boulevard
San Diego, California 92152

Dr. William Woods — 1 copy
BBN
50 Moulton Street
Cambridge, MA 02138

Professor Van Dam — 1 copy
Dept. of Computer Science
Brown University
Providence, RI 02912

Professor Eugene Charniak — 1 copy
Dept. of Computer Science
Brown University
Providence, RI 02912

Professor Robert Wilensky — 1 copy
Univ. of California
Elec. Engr. and Computer Science
Berkeley, CA 94707

Professor Allen Newell — 1 copy
Dept. of Computer Science
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Professor David Waltz — 1 copy
Univ. of Ill at Urbana-Champaign
Coordinated Science Lab
Urbana, IL 61801

Professor Patrick Winston — 1 copy
MIT
545 Technology Square
Cambridge, MA 02139

Professor Marvin Minsky — 1 copy
MIT
545 Technology Square
Cambridge, MA 02139

Professor Negroponte — 1 copy
MIT
545 Technology Square
Cambridge, MA 02139

Professor Jerome Feldman — 1 copy
Univ. of Rochester
Dept. of Computer Science
Rochester, NY 14627

Dr. Nils Nilsson — 1 copy
Stanford Research Institute
Menlo Park, CA 94025

Dr. Alan Meyrowitz — 1 copy
Office of Naval Research
Code 437
800 N. Quincy Street
Arlington, VA 22217

Dr. Edward Shortliffe — 1 copy
Stanford University
MYCIN Project TC-117
Stanford Univ. Medical Center
Stanford, CA 94305

Dr. Douglas Lenat — 1 copy
Stanford University
Computer Science Department
Stanford, CA 94305

Dr. M.C. Harrison — 1 copy
Courant Institute Mathematical Science
New York University
New York, NY 10012

Dr. Morgan — 1 copy
University of Pennsylvania
Dept. of Computer Science & Info. Sci.
Philadelphia, PA 19104

Mr. Fred M. Griffee — 1 copy
Technical Advisor C3 Division
Marine Corps Development
and Education Command
Quantico, VA 22134

Table of Contents

List of Figures

Abstract

This paper descibes a mechanism for dealing with the representation of events and their effects occurring in and over time. The mechanism, which I refer to as a time map manager (TMM), is shown to be useful in problem solvers requiring an ability to reason about time and causality. In addition to describing the theory and its implementation I will demonstrate a programming technique and related discipline based upon the use of data dependencies [Doyle 79a]. This technique supports the design of complex control structures capable of recording the conditions under which information is stored and subsequently responding in highly directed ways when those conditions are changed.

1. Introduction

Most interesting planning problems involve dynamic issues:

- the world changes (and one would hope that our view of it is modified accordingly)
- plans evolve and goals shift to meet our varying desires and aspirations
- execution deadlines approach (and sometimes are ignored for one reason or another)
- assumptions made at one point during planning are invalidated by new knowledge or by constraints imposed by other plans

Each of these problems require an ability to reason about time. A pragmatic foundation for reasoning about time should be able to capture the notion that actions can modify the persistence of facts which are true in the world. It should be able to use information about the duration and sequence of events so as to infer possible effects. Finally it should provide a representational structure to support planning and reasoning about the causal structure of the surrounding environment.

A good deal of energy has been spent on developing logics of time and representations for actions and events which attempt to capture our intuitions about cause and effect [McDermott 82] [McDermott 78] [Allen 81a] [Allen 81b]. The objective is to build a naive but pragmatic theory of causality in order to model the world changing about us and our interaction with it. Using such a model a robot should be capable of developing plans to achieve its goals, executing these plans and recovering from the inevitable problems that arise. To illustrate I'll describe two

simple problems from the task domain which has motivated much of the current research. The domain involves a simple mobile robot, which you can think of as an automated forklift truck, and an environment resembling an industrial machine shop or warehouse.

For the first example suppose that the forklift has two outstanding tasks:

- stack all similar unused items (presumably to conserve floor space) and
- clear all unused items obstructing major thoroughfares in the work space.

(These "tasks" are more on the order of general *policies* [McDermott 77] but they will suffice for the purposes of the example). There are obvious constraints that one could suggest concerning the order in which plans to achieve these tasks are executed. In particular if, say, a hall was cluttered with vacant desks and the forklift was capable of lifting just one desk at a time then certainly it should clear the desks from the hall before stacking other desks upon them. It would also be reasonable to expect it to integrate stacking with clearing where possible.

As a second example consider that the forklift is given the task to transfer an item from one location to another. However let us also suppose that it has been remiss in performing the tasks of the first example and in the course of actually moving the item it encounters an obstacle. The forklift can either change its route (if possible) or set its load down and remove the obstacle before continuing with the transfer task. We might even expect the robot to take the opportunity to satisfy the requirement mentioned in the first example (clearing obstructed thoroughfares) by moving the obstacle to some less traveled area. The latter option involves dynamically integrating new tasks into the flow of planning and execution.

I have built a planning system which deals with the above sorts of problems and a mechanism (TMM) for maintaining temporal maps which supports the necessary reasoning. The rest of this paper will attempt to describe this mechanism and its potential uses in robot problem solving and understanding in general.

1.1 Organization

The structure of this paper is designed to anticipate the interests and possible questions of someone interested in planning programs and planning in general. The chapters on time maps and planning combine theory and implementation. The justification being that the abstraction behind the implementation, data dependency, is closely tied to the theoretical issues being addressed, in particular to the notion of temporal dependence. These chapters tend to be rather technical though numerous examples are included to help motivate the discussion. The last two chapters are more speculative. They are intended to encourage further discussion of the role of time maps in particular and reasoning about time in general.

The basic organization is as follows:

1. introduction
2. time maps
    - what they are
    - some historical perspective on approaches and applications
    - the abstraction
    - the mechanism
3. planning
    - maintaining planning assumptions
    - execution monitoring
4. extended applications in planning
    - continuous planning and execution
    - coordinating a world model with the world it presumes to model
5. conclusions and research directions

2. Time Maps

2.1 What is a time map manager?

Events occur in time. The sort of events we will be investigating, events involving actions, can also be said to occur over time. It is convenient to associate with a particular occurrence of an event a temporal interval demarcating its beginning and ending, in which the event is said to occur. In the following we will refer to such an event occurence as an event token. An event type on the other hand refers to a description of an event without reference to a particular time or occurrence fitting that description (e.g. "an attempt made on the president's life" describes an event type while "John Wilkes Booth shot Abraham Lincoln on April 14, 1865" refers to a specific event token). Event tokens can be compared temporally by means of the relative positions of their beginning and ending points in a single dimensional space, that of time. From the information that two event tokens meet, overlap, or that one occurs during the other it is often possible to infer a causal relationship between them or, in the case that a causal relationship is already known, deduce possible events to follow. For instance knowing that famines follow droughts might lead one to suspect a causal correlation between the two events; knowing that during a cease fire in Lebanon a Shiite Moslem leader was assasinated at the negotiations table one might guess that hostilities would be renewed. Most causal relationships between events imply some sort of temporal relationship as well.

In addition to events causing other events, events can be thought of as causing certain facts to "become true". Each fact caused by a given event is associated with an uninterupted temporal interval designating (1) the point in time at which the event ostensibly made the fact true (it might have already been true) and (2) the first point in time following (1) at which the fact is known to be false. The event is said to enable the fact to persist over a particular temporal

interval which we refer to as a fact token. The beginning and ending of a fact token is often of interest. The fact token spanning this moment and asserting my continued sanity is hopefully not going to end soon. Other events can shorten the duration of a fact token. I might open a window causing a draft because I am warm and someone else might immediately close that window complaining of the cold.

One can hypothesize about a given event occurring at any point in time but the temporal placement of a given event affects both the persistence of the facts caused by that event and the persistence of facts caused by other events. Suppose that we are considering the event *TOKEN1* as occurring at a particular point in time. Facts caused by *TOKEN1* affect the persistence of facts caused by events which precede *TOKEN1* and the facts caused by events which follow *TOKEN1* affect the persistence of facts caused by *TOKEN1*.

A planner must be able to reason about how an event will unfold at a particular point in time. In forming a hypothesis about how an event will occur it is necessary to make certain assumptions concerning the temporal context in which the event is to be placed (i.e. the facts which can be said to be true in adjacent temporal intervals). An event *projection* refers to a set of inferences made assuming that an event will occur in a particular interval. A *projection* of an event token E includes:

1. a set of fact tokens $\{F_1, F_2 \ldots F_n\}$ representing the effects of E
2. a set of event tokens $\{E_1, E_2 \ldots E_m\}$ suggested by causal inference rules to occur given the immediate temporal context of E
3. constraints upon the time of occurrence of the tokens in the sets mentioned in (1) and (2) relative to E
4. ***projections*** of the event tokens in $\{E_1, E_2 \ldots E_m\}$

I may form a projection (or hypothesis concerning how the future is to unfold) based upon certain fact tokens and my current estimates concerning their duration or persistence. Later I may feel obliged to retract my hypothesis upon learning that one of the fact tokens on which I

was depending has been truncated by another event. After opening the window I might expect the room will soon be cool and the air freshened. When the window is shut after so brief a time I must either reconcile myself with the stale air (and my unaccommodating roommate) or do something to restore the flow of fresh air.

A time map manager (TMM) keeps track of the known relative positions of points corresponding to the beginning and ending of fact tokens and event tokens. The TMM maintains a consistent database of fact tokens by limiting the duration of fact token intervals (figure 2-1 illustrates the window example). By consistent we simply mean that no fact token asserting P overlaps a fact token asserting $\sim$P. The TMM allows other programs, under a wide variety conditions, to keep track of the validity of assumptions made on the basis of facts believed to persist over time. For example the assertion that a particular event has occurred can be withdrawn along with its current projection by simply withdrawing support for the assertion. Any other assertions which depended upon this event or any of its derived effects occurring will be called into question as a side effect. In practice this might mean that an "assumption failure" message would be generated annotating the source of the failure and the parties involved. If the occurrence of the withdrawn event is under the control of the program (e.g. the event refers to a task that the program considered executing) and the cost associated with that event not occurring is considerable, then the original state of the data base can be restored by simply restoring the initial event support or by providing new support (i.e. reestablishing the task).

It is also quite easy to displace an event in time while maintaining the temporal relationships between the event and the tokens in its associated projection. This means that a cascade of causally connected tokens can easily be hypothesized to occur at different points in time. Shifting an event from the point in time in which its current projection was formulated to some other point in time may threaten the validity of that projection. For example I may have

assumed that the circus clown on the trapeze would be hurt by a fall but then I discover that the clown's act will follow the high wire act prior to which a safety net will be set up under the trapeze. Such a shift can also threaten other projections. In the chapter on planning (chapter 3) I will demonstrate how the TMM enables a program to keep track of the validity of projections under the above sorts of transformations.

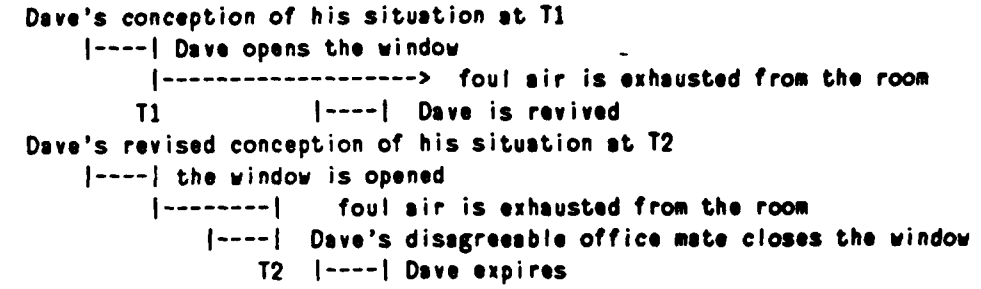


Figure 2-1: A conflict in priorities

In the following we will examine some classic approaches to dealing with events occurring in time and what problems they did and did not solve. Then we will explore the time map proposal and demonstrate how it both subsumes the functions of the earlier approaches and extends what can be expected of a computational method for dealing with events and actions occurring in time. Most of the discussion will center about planning programs though there is a great deal of related work in story comprehension [Charniak 81] [Rieger 76] and text comprehension in general [Allen 80].

2.2 Some Earlier Approaches

Probably the two most discussed planning programs of the last decade are Sacerdoti's NOAH [Sacerdoti 77] and Sussman's HACKER [Sussman 75]. Neither NOAH nor HACKER emphasized an approach to representing time but both were concerned with planning a sequence of actions for accomplishing a task and hence both necessarily had to have some way of dealing with interactions which might arise in executing a proposed sequence of actions. Each program

employed a different mechanism for detecting interactions between plans. I'll briefly discuss each in turn.

Both programs required a data structure in which to represent the changing states of the world over a period of time in which the program proposed to perform certain actions. The object was to record in this structure the effects and preconditions associated with a given action in such a way that subsequent changes to that structure which violated those preconditions or made their effects redundant would be noticed, and appropriate modifications made. I like to think of it in slightly different terms. When we choose a plan we normally do so taking into account the temporal context in which it is to be executed. Our belief that the plan we have chosen will continue to be reasonable is based on assumptions that certain facts which were believed to be true throughout certain temporal intervals will remain so in spite of our changing conception of what the future holds in store. From this perspective a plan interaction occurs when a plan choice assumption is invalidated. Some types of interactions are inconsequential and can be ignored while others may threaten the viability of a particular plan in a given context. When one plan threatens another we say that the two are in conflict. In more general terms we can speak of conflicts between two event projections (or conjectures as to when and how a particular event will manifest itself). In the following I will focus on one rather general class of conflicts and begin by speaking about Sacerdoti's NOAH in order to use his description of such conflicts.

2.2.1 Procedural Nets

NOAH concerned itself primarily in dealing with interactions between actions (an action is represented in STRIPS format [Fikes 71] as a set of preconditions, a list of "facts" which are assumed to be true after the action is performed (procedurally this means a list of assertions to be added to the data base) and a list of "facts" assumed to be false following the action (assertions to be deleted from the data base)). Planning proceeds by filling in the details of some

initial, usually unordered, set of goals or actions to be performed. The current state of the plan is represented in what Sacerdoti calls a *procedural net*; a data structure used for recording the effects of actions and the ordering constraints imposed on their execution. Each action is represented as a node in the net and two actions which are unconstrained relative to one another will appear as child nodes of a single conjunctive node. NOAH delays constraining the execution ordering as long as possible in order to facilitate dealing with interactions. The actions which correspond to the children of a conjunctive node in the *procedural* net can presumably be executed in any order or in parallel if processors, actuators or agents exist for each individual action. Filling in the details of a plan (expansion) consists of expanding all action nodes resulting from the last expansion or all of the initial action nodes in the base case. A node representing an action A at one level of description is expanded by adding new nodes to the net which represent those (sub)actions $\{A_1, A_2, \ldots A_n\}$ characterizing A at some more detailed level of description. Sacerdoti describes [Sacerdoti 77] two conflict types:

> "If an action in one conjunct deletes an expression that is a precondition for a purpose in another conjunct, then a conflict has occurred ."(pg 30)
>
> "Another type of conflict occurs if an action deletes an expression that is a precondition for a subsequent purpose." (pg 31)

NOTE: The expansion of an action A, for our purposes at least, can be thought of as a list of (sub)actions $\{A_1, A_2, \ldots A_n\}$. The last item on this list has particular significance to NOAH. It serves to define the notions of *purpose* and *precondition*:

"The system assumes that the purpose of every action but the last in such an expansion is to establish the truth of some expression in order to make the final action applicable. We will call both the expressions that are to be made true and the nodes that make them true *preconditions*. We will call both the last node in a more detailed expansion of a node and the pattern associated with the last node the *purpose* of the preconditions." (pg 11)

In order to understand what it means for an "action to delete an expression that is a precondition" for another purpose it is necessary to understand how NOAH simulates the execution of a plan during its expansion. During an expansion each expression which is either added or deleted is entered in a data structure called a TOME (for Table Of Multiple Effects)

along with a list of nodes corresponding to the actions which made reference to the expression and whether they added or deleted it. From this table all interactions (i.e. one node adds what another one deletes) are extracted and all expected interactions eliminated. An example of an expected interaction is one in which an action denies its own precondition (e.g. the precondition for *(on A B)* is *(cleartop B)* but executing *(on A B)* causes *(cleartop B)* to be deleted). Those interactions which remain constitute conflicts and are dealt with by either constraining the nodes (i.e. linearization) to eliminate the conflict if possible or failing in that (the nodes are already constrained as in the second case above) splicing in a new node corresponding to an action to reestablish the precondition.

When a conflict between two unconstrained nodes is detected their respective actions are ordered to eliminate the conflict and the structure of the procedural net is modified to reflect the new (partial) execution order. This action is essentially irreversible as NOAH does not retain any information to guide backtracking and as a result fails in finding solutions to certain classes of problems (see [Tate 77] for an account). A later program NONLIN [Tate 75] modeled after NOAH corrected this deficiency by retaining state information from decision points. However in NONLIN backtracking requires restoring the state at the decision point and discarding all other planning knowledge leading up to irreconcileable conflict some of which is likely to still be applicable. The alternative would be to shift the whole node to a different point in the procedural net: a process which cannot easily be done in NOAH or NONLIN except by removing the node, reinserting it at the new spot and then re-expanding it from scratch.

Another problem with the procedural net as a representation of actions in time stems from its poor temporal resolving power. An action in NOAH is essentially treated as a point in time. It is impossible, for instance, to represent a precondition which must be true at the beginning of an action but not necessarily at the end (e.g. for a train to pass successfully through a mountain

tunnel the entrance must be clear when it enters but not necessarily when it leaves). In practice it is always possible to introduce preliminary (gating) actions (e.g. entering the tunnel) to add more resolution to the net, though the notation can become clumsy at times (see the references on temporal logic for more discussion [Allen 81b] [McDermott 82] and the Hendrix article [Hendrix 73] for a proposal to handle time in a more detailed fashion within the STRIPS paradigm).

The STRIPS representation of action also leaves much to be desired. An action can cause other events under the right circumstances and these in turn cause other events in a cascade of effects resulting from a single action. Actions can also cause various facts to persist throughout some temporal interval contingent upon other facts being true. Flicking the light switch will have the consequence of illuminating the room if the switch is functioning and the power is on and any number of other conditions are met. This sort of conditional effect is not supported in NOAH though one can see means of extending the program to incorporate such a facility. A number of more promising approaches are outlined in the references mentioned above on temporal logic.

A final complaint has to do with the procedural net and the nature of planning itself. Planning is a continuous process. Performing an action may cause effects which, if noticed, will require the formulation of new tasks. If I paint the ceiling I could spatter the walls; so I might apply new wallpaper and in the process ruin the floor finish; so I varnish the floor but now my furniture looks so shabby in comparison ... and so on ad infinitum. Moreover there are likely to be tasks which cannot be planned for until after some other task is executed. I won't know what color to paint the trim until I see what the walls look like painted fuchsia. NOAH treats a set of tasks and the plan for achieving them as an isolated process. It has only limited ability to recover from planning errors noticed during execution and it makes no attempt integrate tasks which seek out information from the world in order to direct subsequent planning. A realistic planner

takes into account the limitations of its model of the world; continually supplementing and corroborating its picture of the environment. In the following chapters I will attempt to demonstrate how the approach described in this paper supports continuous planning in the context of a changing representation of the world and the events occurring in it. First let us look at one other planning program and its pragmatic approach to dealing with sequences of actions.

2.2.2 HACKER's Protection Mechanism

Sussman's HACKER took almost the complete opposite approach to planning from that of NOAH. HACKER begins with a completely ordered sequence of actions which constitutes a specification at some level of description of a plan for achieving a task or conjunction of tasks. The specification may turn out to be full of "bugs" or interactions which might compromise the plan but the process of planning consists, in HACKER's view at least, of detailing and *debugging* the initial specification. As with NOAH our treatment here is necessarily brief and superficial. Both NOAH and HACKER are thoroughly described in well written and widely available books [Sacerdoti 77] [Sussman 75]. The purpose here is to point out certain concepts that a planner which reasons about time must address and give some perspective on where those ideas originated. In trying to get some grasp of how these ideas have been incorporated in past programs we will hopefully gain some insight into how they might be incorporated into a more comprehensive and realistic theory of reasoning about time and action.

In this regard we will be concentrating on Sussman's notion of a *protection violation* and the associated mechanism for detecting such violations. The principal ideas had been knocking about in computer science for some time. In Sussman's words [Sussman 75] the "protection mechanism" combines:

> "the ideas of Intentions and Monitors [Hewitt 71] under the unifying concept of the "chronological [vs the lexical or dynamic] scope" of a goal." (my parentheticals) (pg 101)

in order to capture certain intuitions concerning the implied purpose of a goal and implement a

means of recognizing when such purposes are thwarted. The idea is that if an action relies upon a fact being true during some part of the plan's execution then the planner attempts to protect that fact by watching for other actions which make it false within the critical part. Its not clear just how general the actual mechanism used in HACKER was. When watching for *protection violations* HACKER runs in a "careful" mode of program execution in which:

> "whenever a change is made to the world model (the chronological context) ..... all currently protected expressions are fetched from the current dynamic context and tested in the world model. An error results if one is found to be untrue."

When a protected expression is violated an attempt is made to reorder the action associated with the purpose depending upon the protected expression in such a way as to avoid the violation. This detection mechanism is similar in effect to that used by NOAH but there are some subtle differences.

NOAH relied upon a simple syntactic method for distinguishing purpose and precondition. It is clear however that a particular task can have more than one purpose (e.g. cleaning the kitchen will get the landlord off your back and temporarily herd the roaches into the next apartment). The important concept to be learned from HACKER with its more flexible approach to purposes and goals is that a pair of tasks one of which serves the purpose of the other define a temporal interval in which the object of of the servant task is implicitly protected.

The rest of this paper describes an approach to planning involving time maps which subsumes both HACKER's *protection mechanism* and NOAH's procedural net. This approach supports the use of programs (*critics*) such as those used in HACKER and NOAH for constraining the order in which tasks are executed and avoids several of the deficiencies mentioned in the discussion of NOAH.

2.3 Dependency Directed Programming

In order that much of the following discusion be intelligible the reader should be at least passingly familiar with the concept of data dependency [Doyle 79b]. A data dependency network can be thought of as a graph structure whose nodes correspond to beliefs and whose links define support or justificatory relationships. The nodes themselves (which we refer to as ddnodes for data dependency nodes) might be associated with just about anything but for our present purposes suppose that each ddnode is associated with a predicate calculus formula (an assertion or inference rule). In the following we will often refer to the ddnode and its associated formula interchangeably. A ddnode is said to be IN (in our case the associated formula is believed to be true) if there exists a well founded or non-circular justification whose associated ddnodes are themselves IN, or OUT, depending upon the type of support relationships involved. A justification can depend upon some ddnodes being OUT as well as IN. Otherwise a ddnode is said to be OUT. A formula can be made IN by simply asserting it, in which case its justification is the fact that it is a premiss. In other cases a formula can be IN or OUT depending upon the current status of other formulas in the network.

One important fact about data dependency mechanisms (at least those modeled after Doyle's truth maintenance system) is that they support non-monotonic inference. In data dependency terms, this means that changing the status of an existing ddnode can cause a change in the status of other ddnodes. In particular asserting a new premiss can cause something that was formerly IN to become OUT. For example suppose that my belief that stockpiling nuclear weapons is an effective deterrent to their use is contingent upon my belief that the Soviets fear our nuclear capability. If I later learn that the Soviets think our retaliatory capabilities can be effectively disabled by their first strike offensive then I might wish to reassess more than a few of my beliefs.

Since a data dependency system retains OUT justifications for beliefs (unless explicitly told to

excise them) a particular assertion can toggle between IN and OUT depending upon the context in which it appears (i.e. the status of other formulas in the network). The notion of context has been generalized to mean a set of formulas and a status assignment for that set. Thus a given formula may appear IN in one context but OUT in another. Later we will discuss a mechanism for switching efficiently between contexts [McDermott 81] which allows a planner to entertain multiple hypotheses concerning possible futures. For the time being this should suffice for understanding the discussion though it should be noted that data dependence networks form the basis for a powerful abstraction which has only been hinted at in the preceding paragraphs and the uninformed reader is urged to educate himself more thoroughly. TMM and the programs which rely upon it derive much of their power from control structures directed by data dependence hierarchies.

2.4 Time maps in the abstract: the notion of temporal dependence

The general ideas behind time maps and time map maintenance are quite simple. They rely upon the notion of data dependence. A time map designates a partial order on points. Intervals are described as ordered pairs of points with the stipulation that the first point in a pair precedes the second one. Each pair is associated with a token and certain tokens are distinguished by their designating a (temporally dependent) fact. The maintenance algorithm simply sees to it that if there exists a pair (BEGIN1 END1) referring to a fact P, then for all pairs (BEGIN2 END2) referring to $\sim$P it is not the case that either BEGIN2 $<$ BEGIN1 $<$ END2 or BEGIN1 $<$ BEGIN2 $<$ END1. When such an overlap is detected the token occurring earlier is shortened so that it ends before the later token begins.

If facts were added to the time map and never erased or moved about then enforcing this invariant would be simple. The problem is that we are anticipating the needs of a planning algorithm which will use the time map, and those needs dictate the flexibility to quickly add,

shift, remove and restore arbitrarily large event projections. Suppose for example someone tells you that Tycho Brahe took into account dopler red shift in his tables plotting the motions of celestial bodies or perhaps that tomorrow you will be required to give a lecture at 9:00 AM though you had planned to sleep late and play squash. In the first case your belief in when the red shift phenomena was first understood should make it difficult to assimilate the Tycho Brahe story and in the second you might either modify your projected schedule or protest the requirement. In order to achieve flexibility the maintenance system must keep track for each fact token asserting P, those fact tokens asserting $\sim P$ that are "likely" to change position in the temporal partial order.

Determining a reasonable context of events for planning or other forms of problem solving is often a significant part of the problem. Suppose that I'm trying to reconstruct yesterday's events during which my office was entered and my coffee cup with the broken handle and chipped lip stolen. I might be justifiably interested in the event in which I went to the cafeteria, carelessly leaving the office door unlocked behind me. But its not likely that the events corresponding to my waking yesterday morning or riding the shuttle last night will be of any use in discovering just who might have perpetrated the crime.

A practical TMM might designate temporal windows or a set of categories for determining the sort of events that might be considered valuable during planning. Unfortunately fixed-duration windows are likely to prove too restrictive and it is difficult to specify reasonable categories that capture what we mean by relative importance. The duration of an event is certainly not a reliable indicator: the bomb detonations at Hiroshima and Nagasaki together spanned at most a few milliseconds though their repercussions are likely to extend well into the next millenia (if we're so lucky). On the other hand the duration of an event's effects is no indication either, otherwise every person's death would would be considered as an event on a cosmic scale. Rather

than introduce events on the basis of some inherent property of the events themselves it seems more reasonable to leave it up to the planner to establish criteria for inclusion.

The solution that I have adopted is to keep track of a map *kernel*, the set of events that the TMM is currently operating on. It is assumed that any events which the planning program deems relevant will be introduced into this kernel. As long as an event remains active in the kernel its projected effects will be taken into account by the consistency maintenance algorithm. In the planner we will be discussing later in this paper, the kernel consists only of tokens corresponding to tasks or events which have yet to occur or be executed and facts whose tokens persist at least into the immediate future and upon which some currently active task has based its choice of a plan for carrying out that task.

At this point it is obvious that knowledge about events occurring in time can become available in at least two forms in the data base. There are the privileged few events currently residing within the kernel: privileged in the sense that these events, in isolation at least, present to the planner a consistent world model. Then there are all those events that we are not currently concerned with and among which we might find inconsistent temporal assertions. This introduces two additional problems which we will briefly discuss and then introduce simple interim solutions which have been adopted in the working version of the TMM-based planner.

The first problem has to do with the status of events lying outside the kernel. How are facts caused by events no longer in the kernel retrieved efficiently from memory? The answer is, that they may not be efficiently retrieved. They will have to be searched for and how efficient that search is will depend upon whatever indices are currently available (perhaps through those events currently residing in the kernel). It is up to the planner to index the events relative to other events in such a way that given one event E1 it is reasonably simple to find events whose effects may influence our consideration of E1. This problem is of great interest but orthogonal in most

respects to the problem of maintaining time maps. In order to simplify the discussion and maintain some common ground between the planner described here and the planners with which it is compared we have adopted a plan and event description syntax not unlike Sacerdoti's SOUP. No attempt will be made in this paper to describe the process of assimilating knowledge garnered in the process of planning and execution into a well indexed and integrated episodic memory.

The second problem introduced by a privileged kernel has to do with removing items from the kernel which are no longer relevant to the planner's immediately active tasks. The routine which performs the removal of a token from the kernel has to perform some rather complex juggling in order to maintain all active assumptions in the network. An event token must be disentangled from its projection in such a way as to maintain dependency relationships and yet reduce the size of the kernel. These token removal routines will be described in some detail later in the paper (section 4.2) along with a discussion of criteria for including events in (and subsequently removing them from) the kernel (criteria which might support other uses of the TMM; say for example story comprehension).

2.5 A description of the actual TMM algorithms

In the following I will assume familiarity with some sort of deductive retrieval mechanism (PLANNER type languages, PROLOG, QA4 or the like). The actual system we employ is called DUCK [McDermott 81] and is in many respects similar to AMORD [deKleer 77] in that it integrates the basic retrieval mechanisms found in CONNIVER [McDermott 73] with a data dependency mechanism for maintaining support relationships.

A few functions found in DUCK which we will make frequent reference to will need a bit of explanation. *(Fetch pattern)* returns a generated list of all assertions in the data base which match *pattern* or which can be derived from rules and assertions in the data base. If no

assertions can be found then *fetch* returns nil. *(Premise pattern)* adds to the data base the assertion got from *pattern* by binding its free variables (of the form ?var-name) in the obvious (lexically scoped) manner. *(ans-support <list of supports> <things to do>)* for our purposes simply ensures that assertions added within the dynamic scope of <things to do> will be dependent upon the support conditions in <list of supports>, where supports are of the form *([+|-] [pattern|pattern-list])*. "+" means depends (positively) on the associated assertion(s) being IN and "-" on it(their) being OUT. Finally *(for-each-ans (fetch pattern) <things to do>)* is a looping construct which is best thought of as code which expands into:

```
(for (assertion-returned in (fetch pattern))
     (ans-support (+ assertion-returned)
             <things to do>))
```

Wherever convenient, to keep things simple, I will deviate from the actual syntax of DUCK in describing algorithms.

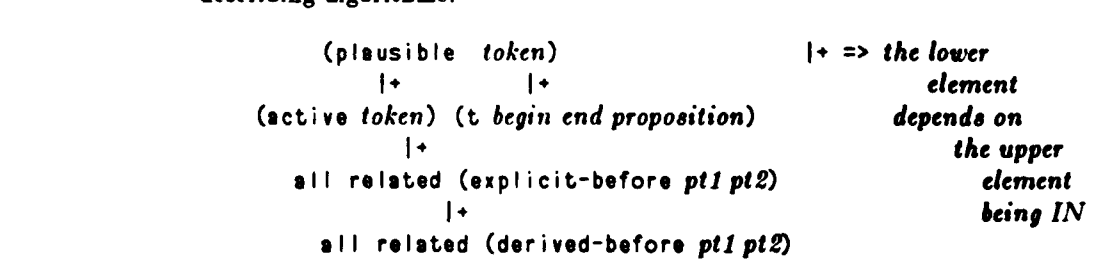


Figure 2-2: Part of the TMM dependency hierarchy

As we mentioned a great deal of the power of TMM and the systems for detecting assumption failures which we will discuss later comes from the use of dependency hierarchies. Some of the dependencies used in the TMM are shown in figure 2-2. The assertion predicates used in the diagram require a bit of explanation. A fact token referring to P is associated with a *tassertion* in the data base of the form *(t begin end P)* where *begin* and *end* correspond to the beginning and ending of the token interval. A token is *plausible* if it is believed to have occurred or to be going to occur. A token is *active* if it is in the kernel of the TMM. A *tassertion* is a kernel data structure: its status is dependent upon its associated fact token being *active*.

All internal temporal relationships are represented by the *derived-before* predicate. The efficent operation of the TMM consistency algorithm relies upon the explicit presence in the data base of the transitive closure of *derived-before* on the set of all begin and end points of tokens in the kernel. When the planner introduces a constraint upon two tokens in the kernel (e.g. that one token begins before the other) then appropriate *explicit-before* relations are added to the data base dependent upon the two tokens being active and whatever other reasons the planner has for adding this constraint. At the same time the transitive closure on the kernel points is updated so that all resulting *derived-before* assertions are dependent upon the *explicit-before* assertion which gave rise to them. The *explicit-before* assertions remain in the data base as long as the related tokens are *plausible* but the *derived-before* assertions are part of the kernel and hence they exist only as long as the related tokens are *active*.

```
(define update-before-tm (pt1 pt2)
   (cond ((fetch (derived-before pt1 pt2)))
         ((fetch (derived-before pt2 pt1)) 'INCONSISTENT)
         (t (premiss (explicit-before pt1 pt2))
            (ans-support (+ (explicit-before pt1 pt2))
                   (premiss (derived-before pt1 pt2))
                   (for-each-ans (fetch (derived-before pt2 ?ptA))
                       (premiss (derived-before pt1 ?ptA)))
                   (for-each-ans (fetch (derived-before ?ptA pt1))
                       (premiss (derived-before ?ptA pt2))
                       (for-each-ans (fetch (derived-before pt2 ?ptB))
                            (premiss (derived-before ?ptA ?ptB))))))))
```

Figure 2-3: An algorithm for updating the *derived-before* transitive closure

The algorithm (see figure 2-3) which maintains the transitive closure is a simple graph algorithm. It is only of interest here to demonstrate how the dependencies are introduced so as to ensure that when further additions and deletions are made to the data base only the correct subset of *derived-before* assertions will remain.

Now we can describe the *tassertion* consistency algorithm. A *tassertion* is added to the data base dependent upon its associated fact token being *plausible*. As long as the fact token remains

```
(define tassert-tm (fact-token begin1 end1 assertion)
 (let ((neg-assertion (get-assertion-negation-tm assertion)))
      (ans-support (+ (active fact-token))
          (premiss (t begin1 end2 assertion)))
      (ans-support (+ (t begin1 end2 assertion))
        (for-each-ans (fetch (t ?begin2 ?end2 neg-assertion))
          (ans-support ((+ (derived-before begin1 ?begin2))
                        (- (derived-before ?begin2 end1)))
              (premiss (inconsistent-order  ?begin2 end1)))
          (ans-support ((+ (derived-before  ?begin2 begin1))
                        (- (derived-before  begin1 ?end2)))
              (premiss (inconsistent-order  ?begin1 end2)))))))

(rule restore-consistency
      (if-added (inconsistent-order ?point1 ?point2)
                (ignoring  (inconsistent-order ?point1 ?point2)
                    (update-before-tm ?point1 ?point2 ))))
```

NOTE:

i. *Get-assertion-negation-tm P* simply returns *not P* if *P* is not of the form *not P'* and *P'* if it is.

ii. In the context of *ans-support*, *premiss* simply installs a new ddnode contingent upon the status of the support clauses. Hence it is possible that the resulting status of the premissed assertion is OUT though subject to subsequent changes in the status of support clauses.

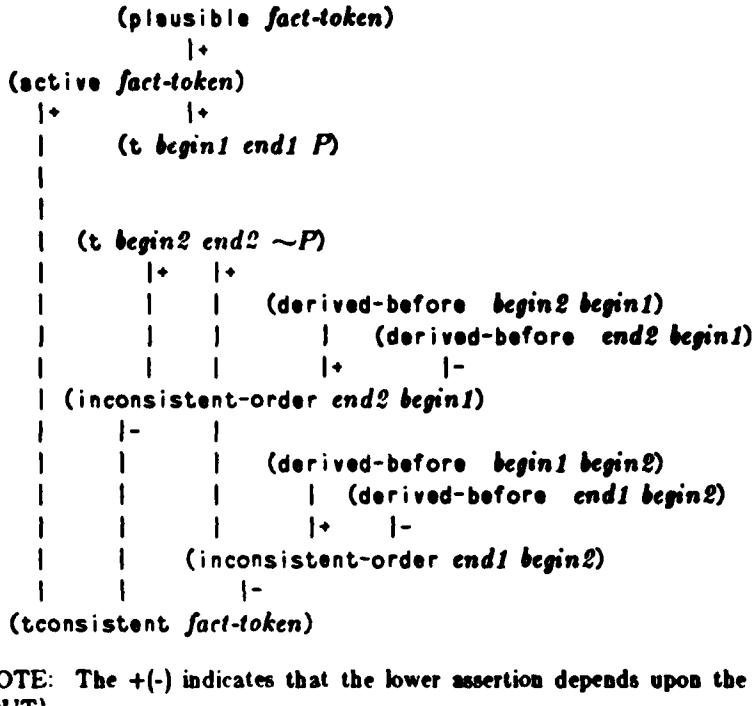
iii. *(If-added pattern things-to-do)* says that if an assertion matching *pattern* becomes IN evaluate the things to do.

iv. All assertions premissed in the dynamic scope of an *if-added* rule are made dependent upon the assertion which triggered the rule being IN . In our case however, it is the purpose of *restore-consistency* to make the triggering assertion OUT. The *ignoring* construct enables us to circumvent the default support machinery.

v. A diagram showing the dependency relations for maintaining kernel consistency is shown in 2-5.

Figure 2-4: Algorithms for adding the new tassertions and resolving inconsistencies

active in the kernel its duration is contingent upon other tokens in the kernel. I will spare the reader the details (the algorithm is straightforward and listed in figure 2-4 for those interested) and supply the basic method of setting up the dependencies and restoring consistency when a dependency failure is noticed. At the time a *tassertion* T1 = *(t begin1 end1 P)* is added a check is made of all currently active *tassertions* of the form T2 = *(t begin2 end2 ~P)*. For each such *tassertion* T2, if *(derived-before begin1 begin2)* is IN then it must also be the case that *(derived-before begin2 end1)* is IN and similarly if *(derived-before begin2 begin1)* is IN



NOTE: The +(-) indicates that the lower assertion depends upon the upper assertion being IN (OUT).

Figure 2-5: Dependency hierarchy for maintaining *tassertion* consistency

(derived-before begin1 end2) must also be IN.

The *tassertion* initialization process *tassert-tm* makes the appropriate changes, if necessary, using update-before-tm (above). It then creates a set of *inconsistent-order* assertions which are initially OUT but whose justifications are set up so as to capture the *derived-before* mandates just stated. Remember that the TMM is really only interested in the persistence of fact tokens: that is to say the position of their endpoints in the time map partial order. The *inconsistent-order* assertions refer to constraints which must be imposed if certain conditions become true.

In order to detect and correct inconsistencies, a forward chaining rule is set up in the data base to call a function to restore consistency whenever an assertion of the form *(inconsistent-order ?point1 ?point2)* changes its status from OUT to IN. Thus whenever a consistency mandate is

violated its associated *inconsistent-order* assertion will become IN and consistency can be restored as during initialization using update-before-tm. With this algorithm a *tassertion* need only be "aware" of *tassertions* added before it, as *tassertions* added afterward will take care of any interactions which they cause.

2.5.1 A Detailed Example

In this section I'll try to present an example of how the TMM routines might be useful in reasoning about events and the persistence of their effects. The following example involves testing the alibi of a suspect implicated in a gothic murder mystery. The scenario is contrived in order that it be both brief and illustrative. I apologize to any disappointed (or offended) gothic mystery afficiandos.

To begin with the events of the interval (EVENT1) surrounding the mysterious death of Sigmund Montcalm IV:

1. Inspector MacPherson from the Yard found Sigmund in the study of the castle at approximately 1:00 AM. The recent heir to the Montcalm estate was pronounced dead by Dr. Corry the police surgeon; Montcalm's death was attributed to a bullet wound in the chest. We'll refer to this as EVENT5.
2. When MacPherson entered the study the lights were on (FACT2) and Sigmund was clutching a copy of Proust's "Remembrance of Things Past" (FACT3). The book was covered with fresh fingerprints determined to belong to Montcalm's aging butler, Jarvis.
3. Investigators found out that the castle security system automatically turns off all lights in the castle (EVENT3) from midnight till 6:00 AM (castles are so expensive to maintain) and only Jarvis had a key which can overide the system.
4. The copy of Proust's book that Sigmund was holding normally resides on a shelf in the study but the upstairs maid said that Jarvis had removed it from the study the previous day (EVENT2).
5. Jarvis maintains that he returned the book (EVENT6) early in the evening before Sigmund was found shot and probably turned on the study lights at that time (EVENT7). He claims to have spent the rest of the night in the company of Margaret Squally enjoying a quiet walk on the Thames.

The inspector wishes to see if Jarvis' alibi is reasonable given the facts. Figure 2-6 shows the initial TMM display (added comments are italicized) with no constraints upon when EVENT6

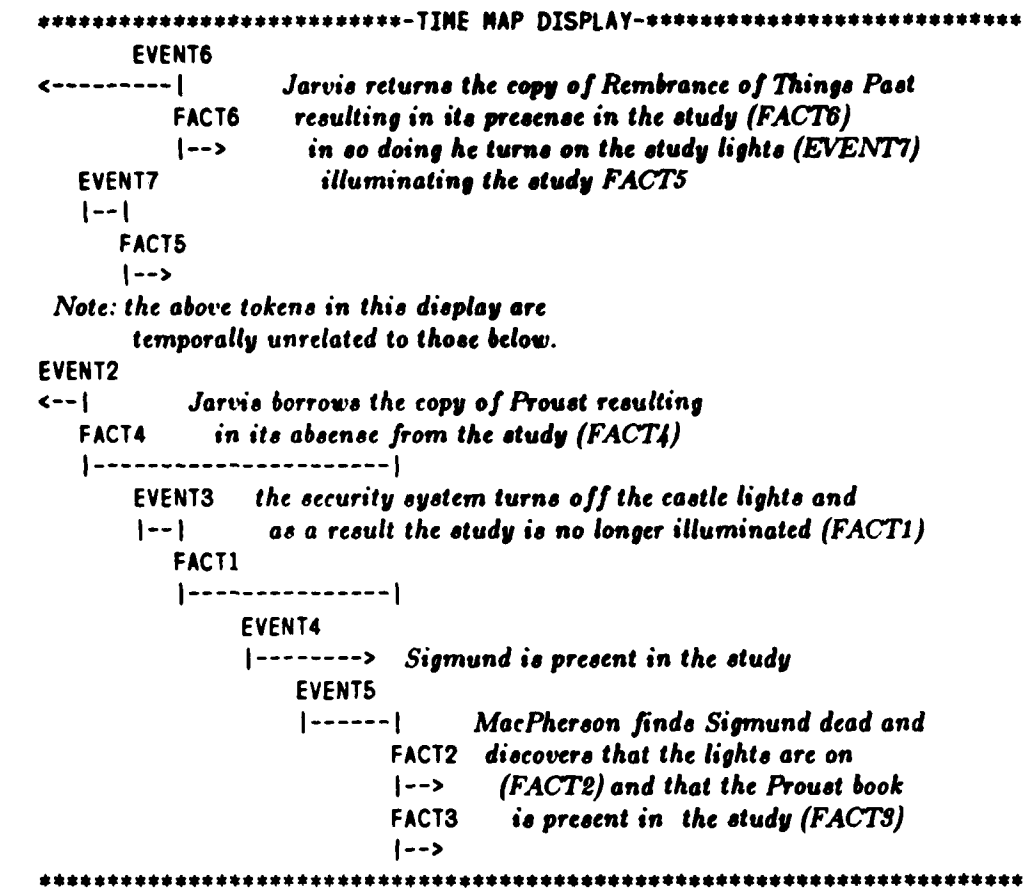


Figure 2-6: The initial scenario

occurred .

If Jarvis is not lying and he did return the book before midnight then it is difficult to explain the fact that the lights were on in the study when the police arrived (figure 2-7 displays the facts as seen from the TMM data base assuming Jarvis is telling the truth).

If on the other hand Jarvis returned the book after midnight then the facts fit together quite nicely. But if this conjecture (see figure 2-8) is correct why did Jarvis mislead the police?

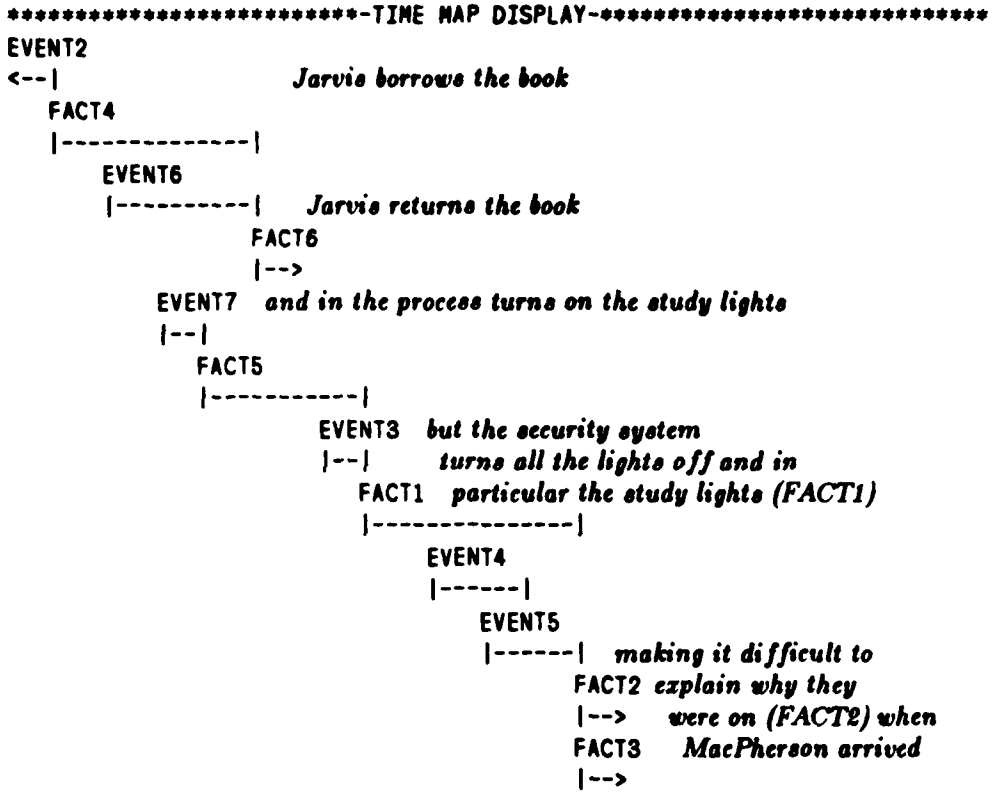


Figure 2-7: Jarvis' alibi

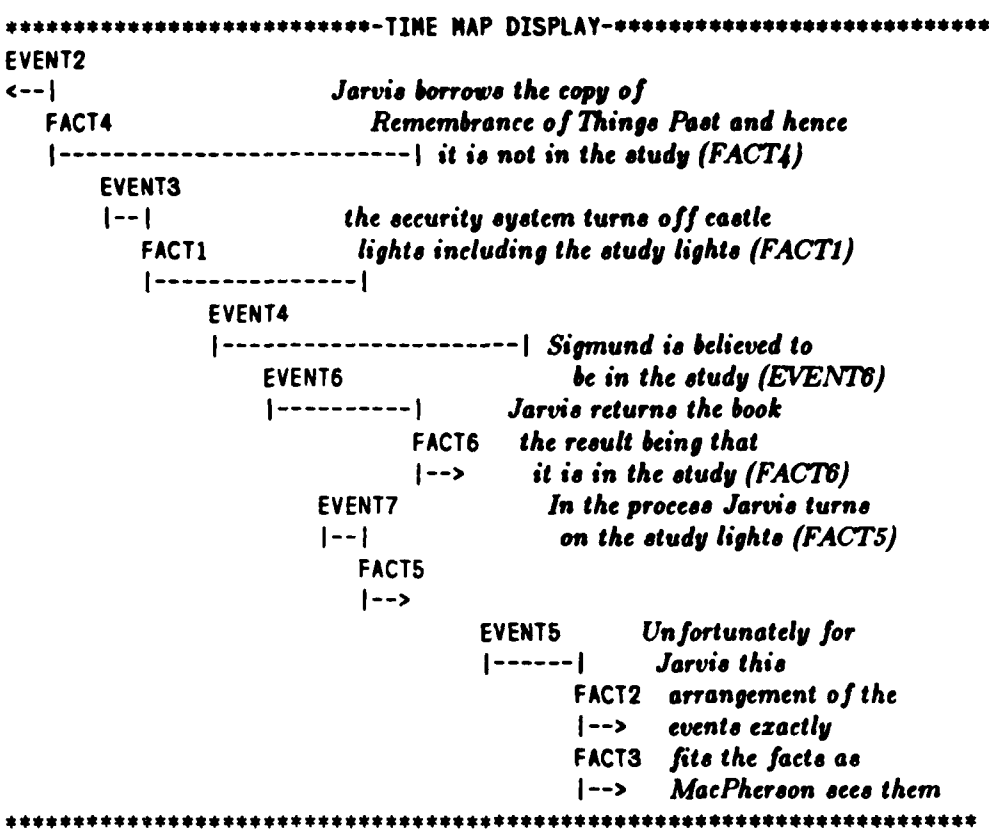


Figure 2-8: Condemning evidence

3. Planning

There are two basic inferential operations that add tokens to the time map kernel. *Projection* is the inference of the likely consequences of an event (e.g. if one country occupies land which another country claims then a conflict is likely to ensue). *Expansion* is the inference of how the event will occur; that is, a description of its subevents and their interrelationships (e.g. an invasion might be expanded into three subevents: massing troops at a strategic border under the pretense of routine maneuvers, occupying certain key points in the coveted territory during a swift "police" action "justified" by national security and finally the eradication of opposition forces and the setting up of permanent garrisons and fortifications). When the event being expanded is a task of the planner, expansion corresponds to adding to the kernel those subtasks associated with a specific plan for achieving that task. This process is generally called task *reduction*.

Temporal elaboration is the process of expanding tokens in order to add precision to the model and projecting token consequences to determine what effect an event is likely to have. There is certainly more to planning than simply this. Plan evaluation, scheduling and what to do when there is no satisfactory plan are all presumably part of planning. But in the following we will focus primarily upon this process of elaboration and the detection of the sorts of interactions between event projections which were discussed in the section on NOAH and HACKER.

3.1 Maintaining Assumptions

Expansion (or plan choice in the case of a task) corresponds to a hypothesis concerning how an event will turn out (or how a task can be best achieved). The choice of expansion for a given situation will depend upon the temporal context in which it is currently assumed to occur. As was mentioned earlier the warrant for continued belief in an expansion is based on assumptions

that certain facts which were believed to be true throughout certain temporal intervals will remain so in spite of our changing conception of what the future holds in store. An *interaction* occurs when a plan choice assumption is invalidated, and a *conflict* is an interaction which threatens an expansion's credibility. In the following sections I will discuss the mechanism whereby a plan is made sensitive to its temporal context, how conflicts are brought to the attention of the planner and how these conflicts can be meaningfully annotated in order to facilitate the task of resolving these conflicts.

A great deal of work has gone into classifying types of "bugs" [Sussman 75] or categories of plan interaction [Sacerdoti 77] [Wilensky 83]. We will concern ourselves here with one general category of conflict whose detection can be efficiently implemented. Later we will discuss a second mechanism, that of monitors [McDermott 77], which can be used to detect a larger class of interactions but at somewhat greater computational cost.

3.1.1 Plan-Expansion-Time Assumptions

Both NOAH and HACKER employed some variant of the STRIPS representation for actions and their effects, involving *preconditions*, *add-lists* and *delete-lists*. For representing events and their possible projections I have borrowed from the logics of action and time of James Allen and Drew McDermott [Allen 81b] [McDermott 82]. The representation attempts to overcome some of the difficulties with the STRIPS approach mentioned earlier. In particular the notation provides a more detailed description of the effects of an event and the underlying logic provides greater resolution for reasoning about time. For representing the effects of an event:

- **event causation:** *(ecause TOK1 TOK2)* denotes that the event token TOK2 will follow the event token TOK1 and that the plausibility of the second occurring is contingent upon the first occurring.
- **fact causation:** *(pcause TOK1 TOK2)* denotes that the fact token TOK2 will follow the event token TOK1 and persist for some indeterminate period and that, as before, the plausibility of the second occurring is contingent upon the first.

Instead of *preconditions* I will refer to (explicit) planning assumptions. The only reason for this

change is that in some cases it seems awkward to call certain plan choice constraints preconditions. For instance I may choose to walk downtown simply because I have no other means of transportation at my disposal. When I am later offered a ride, my plan to walk is no less viable though my choice of plan seems in need of reassessment. The ride offer did not violate a *precondition* for the success of the walk-plan though it could be said to violate a *precondition* for the walk-plan's appropriateness.

Before describing the formal syntax for specifying plan choice criteria let's examine some categories of assumptions that might be employed. First that a temporally dependent fact (a fact token) stands in some relation to the interval in which the task (for which a plan is being sought) is currently scheduled to occur. For example the choice of what plan to use for the forklift loading a tractor/trailer may depend upon how the tractor/trailer is situated relative to the loading dock and whether or not the trailer doors will be open when the plan is executed.

A second category has to do with plan choice criteria which require the absence of adjacent intervals in which a fact is believed to be true. I don't assume that at any point in time either a fact or its negation is believed. Some things the planner simply has no knowledge of and it is often to its advantage to realize the extent of its ignorance. So for example it might be convenient to express the dictum to always load heavy items first (forward toward the cab/tractor thus distributing the load) unless it is known that a particular item is to be unloaded early.

The last category is meant to capture the notion that if, after my initial choice, some fact token fortuitously occurs meeting certain temporal criteria then I would like to be alerted so that I can reconsider my options. Suppose the forklift anticipates the need for an object upon which it can temporarily rest its load while carrying out a task. It may have no knowledge of an unencumbered object meeting its specifications at plan choice time. Instead it chooses some

alternative (and presumably more expensive) plan contingent upon the unavailability of the desired objects. Later, if during its exploits or during subsequent planning it "discovers" an appropriate object available in a surrounding interval, it should be alerted to determine whether or not a "better" plan is applicable. Sacerdoti's *eliminate-redundant-preconditions* critic performs a similar (though less general) function in NOAH.

The syntax for specifying plan choice criteria is shown in figure 3-1.

```
(to-do (some-task ?var_1 . . . ?var_k)
    (use <first-plan-name>
         assuming   <list of conditions which must hold>
         in-lieu-of <list of conditions which must not hold>)
    (use <second-plan-name>
         . . . etc)
    (else-use <default-plan-name>))
```

Figure 3-1: Format for specifying plan choice criteria

Conditions are of the form *(currently-spans <fact>)* or *(currently-meets <fact>)*. The condition *(currently-spans <fact>)*, specified with respect to a task occurring in an interval *I1*, refers to a constraint involving an interval *I2* in which *<fact>* is believed to be true and *I2* begins before and ends after *I1*. (*(currently-meets <fact>)* is similar with the exception that it is only necessary that the *<fact>* be believed at the start of the task). These conditions are expanded to integrate with the time-map machinery forming the basis of the conflict recognition scheme. The order in which plans are considered can be massaged dynamically by using the keyword *option* or *conditional-option* as in *in-lieu-of ((option <test> <plans to be tried first>))*. For instance suppose you have a task to get rich and you have a plan which involves winning the lottery. The method for determining whether the lottery plan is reasonable is rather complex and time consuming (it involves tarot cards and consulting an augury). However you have an indicator (a magic bunion) which, if it responds positively, bodes well for your chances with the lottery and hence warrants your making the more complex applicability tests. This

might be represented as:

```
(to-do (in-the-money ?poor-jerk)
    (use drudge-work in-lieu-of
                    ((option (throbbing-bunion ?poor-jerk) win-big)))
    (use win-big    assuming ((tea-leaves favorable ?poor-jerk)
                              (signs&portents high-times ?poor-jerk)))
    (else-use dream-on)))
```

In the get rich example the *option* clause is only checked at plan choice time. Such an option can also be recalled in the event that it later becomes IN by using an *in-lieu-of* clause of the form *(fortuitously <test> <plans>)* in which case if <test> becomes IN an attempt can be made to see if one of <plans> is applicable. This might mean in the get rich example that if you went to work and suddenly your bunion started acting up you might drop everything and run to the local soothsayer.

To illustrate how plan choice assumptions are used I'll present a simple (if somewhat frivolous) example which demonstrates some of the main features of the plan choice specification format.

Suppose that you wish to return a book, Goethe's Faust, to the library. You have in mind a particular route, namely the scenic path through the park, and a particular mode of transportation, your bicycle. Suppose further that you only ride your bike when you know it will be clear and that after dark the path is closed to bicycles to protect muggers and other people who require those dark avenues to ply their trades. The *transport* task has the following specification:

```
(to-do (transport ?object ?route)
     (use bicycle assuming  ((relatively-small ?object)
                             (currently-spans (clear-skies)))
                  in-lieu-of ((fragile ?object)
                             (currently-meets (closed ?route))))
     (else-use walk))
```

Now let's fill in the context in which plan choice is to occur. The particular *transport* task that we're interested in is associated with the event token EVENT5. Suppose that you have already

planned to listen to the weather on the radio before running your errand and that the event token corresponding to this is EVENT3. You know from last night's news, EVENT2, that the skies should be clear for the next few days. Finally we have the events associated with lunch and dinner, EVENT4 and EVENT6. As the scene unfolds you have already decided to take the book back after lunch and you are well aware that after dinner the path will closed to bicyclists.

The following is a demo of the planner initialized as per the above description. Some of the irrelevant annotations have been deleted and added comments are italicized.

```
-> (test)                                    setting up the scenario
Initializing EVENT2 with status READY        last night's news
Initializing EVENT3 with status READY        listening to the radio
Initializing EVENT4 with status READY        lunch
Initializing EVENT5 with status READY        returning the book
Initializing EVENT6 with status READY        dinner

Initializing FACT1 with status FACT
ADDING (t POINT4 POINT14 (clear-skies))
Initializing FACT2 with status FACT
ADDING (t POINT12 POINT16 (closed bike-path))

  *************************-TIME MAP DISPLAY-*************************
  EVENT2
  <--|        during last night's news (EVENT2) we are told that
     FACT1      clear skies will prevail over the region
     |-->        for the next few days (FACT1)
         EVENT3
         |--|
                 EVENT4              Note:
                 |--|                  the display can be ambiguous
                         EVENT5        at times. EVENT5 and EVENT6
                         |-->           cannot be compared here
                         EVENT6
                         |--|       we know that after dinner(EVENT6)
                            FACT2     that the bike path will be
                            |-->        closed to cyclists (FACT2)
  ************************************************************************

BEGINNING TEST SEQUENCE

Plan chosen in EVENT5 for task (transport Goethes-Faust bike-path)
 is bicycle.
                              Scheduling advice simply notes potential
Scheduling advice:             conflicts and suggests simple remedies.
  ((BEFORE EVENT5 EVENT6)
```

```
   (passume EVENT5 (pout (meets (closed bike-path))) update))

ADDING (PASSUME EVENT5 (pout (meets (closed bike-path))) update)
ADDING (PASSUME EVENT5 (pout (fragile Goethes-Faust)) update)
ADDING (PASSUME EVENT5 (pin (spans (clear-skies))) update)
ADDING (PASSUME EVENT5 (pin (relatively-small Goethes-Faust)) update)

Suppose that while listening to the radio I learn of an
approaching storm, EVENT7, which will hit before noon.

Initializing EVENT7 with status READY
                                        an inference is made
Asserting that EVENT7 causes (not (clear-skies)).

Initializing FACT3 with status FACT
ADDING (t POINT18 POINT20 (not (clear-skies)))
                                    TMM notes an inconsistency ..
POSSIBLE CLASH WITH  (t POINT4 POINT14 (clear-skies))
                                    .. and resolves it by truncating the
RESOLVING BEFORE POINT4 POINT18     persistence of FACT1
                    it appears an assumption is violated
ERASING (PASSUME EVENT5 (pin (spans (clear-skies))) update)
                    an attempt is made to reconfirm the assumption ..
UPDATING (pin (spans (clear-skies))) for EVENT5

Scheduling advice:
  ((BEFORE EVENT5 EVENT7)
   (passume EVENT5 (pin (spans (clear-skies))) update))
                                    .. but the attempt fails
SUSPICIOUS PLAN bicycle -> EVENT5

INTERACTIONS AFFECTING EVENT5:
   ((pin (spans (clear-skies))) update EVENT7 EVENT5)
```

```
**************************-TIME MAP DISPLAY-**************************
EVENT2
<--|
   FACT1
   |--------------|
       EVENT3
       |--|                the storm represented by EVENT7
              EVENT7        causes the sky to cloud (FACT3)
              |--|          thus truncating our belief that it
                 FACT3      would remain clear (FACT1) throughout
                 |-->       our trip to the library (EVENT5)
                      EVENT4  a belief instigated by last night's
                      |--|                  news (EVENT2)
                             EVENT5
                             |-->
                             EVENT6
                             |--|
                                FACT2
                                |-->
**************************************************************************

RESTORING INITIAL STATE  Now we restore the time map to its initial
                         state pretending we never listened to the radio
Now for some reason the trip to the
 library gets delayed til after dinner.

Scheduling EVENT6 before EVENT5
                          another assumption fails this time
ERASING (PASSUME EVENT5 (pout (meets (closed bike-path))) update)

SUSPICIOUS PLAN bicycle -> EVENT5

INTERACTIONS AFFECTING EVENT5:
    ((pout (meets (closed bike-path))) update EVENT6 EVENT5)

**************************-TIME MAP DISPLAY-**************************
EVENT2
<--|
   FACT1
   |-->
       EVENT3
       |--|
              EVENT4
              |--|                    The library trip, EVENT5, is
                             EVENT5   constrained to occur after
                             |-->       dinner, EVENT6, at which time
                      EVENT6            it is known that the path in
                      |--|                the park is closed to
                         FACT2              bicyclists, FACT2
                         |-->
```

```
  ************************************************************************

RESTORING INITIAL STATE  Again we restore the time map to its
                                    initial state
TESTING THE ASSUMPTION UPDATE MECHANISM

Suppose that we learn that the city began
 this morning repaving the bike path, EVENT8.

Initializing EVENT8 with status READY
Initializing FACT4 with status FACT    an inference is made
ADDING (t POINT22 POINT24 (closed bike-path))
     At this point the correct dependencies are in place ..
Scheduling EVENT8 before EVENT3
     .. so that if a compromising constraint is added ..
ERASING (PASSUME EVENT5 (pout (meets (closed bike-path))) update)
     .. an assumption failure is noted as before.
SUSPICIOUS PLAN bicycle -> EVENT5

  *************************-TIME MAP DISPLAY-**************************
        EVENT3
        |--|
           FACT1
           |-->
                EVENT4
                |--|
                        EVENT6
                        |--|
                           FACT2
                           |-->
                        EVENT5
                        |-->
  EVENT8            it is discovered that the city will
  <--|               begin paving the bike path (EVENT8) in
     FACT4           the morning causing it to be closed (FACT4)
     |-->             spoiling the plan in EVENT5.
  ************************************************************************
-> (exit)
```

3.1.2 Problems Involving Classes of Constraints

In stating assumptions concerning the applicability of a particular plan for a given task it often seems that we can completely specify the relevant applicability conditions. People are actually surprised when someone asking directions fails to comprehend a staccato burst of street names and landmarks. This is partly because the direction giver's familiarity with the details leads him

or her to assume that these details are implicit in the directions. I might tell an out-of-town visitor who wishes to do some shopping at Macy's to board the Orange Street bus at the corner of Cold Spring Road and Orange and get off at Chapel Square Mall. To New Haven residents this would be more than sufficient but a visitor is likely to find the description inadequate. Besides assuming a great deal of auxiliary planning knowledge these directions fail to specify a particular bus. What if a Trailways bound for Schenectady happens along? The fact is that I had in mind a particular bus route, and I assumed that the person using the plan would understand the level of the plan specification and fill in the details from that great body of arbitrary assignments, conventions and jungle savvy that we all share.

Had I wanted to describe the plan to someone less proficient in urban orienteering I might have said to get on a bus which stops on Orange Street, is traveling in the direction of downtown, and which has either "Chapel Square Mall" or "Yale Shuttle" displayed over the windscreen on the front of the bus. But even this will be inadequate for someone who does not know what a bus looks like, can't read or is lacking in some other necessary bit of knowledge or cognitive faculty. If on the other hand I was to say to get on the bus we took yesterday then one might assume (incorrectly) that I was referring to a specific bus; one which the intended recipient of the advice is believed to have direct knowledge of (e.g. the one with *$#&% spray painted across the side and the slashed seat cushions). The problem is that sometimes we are referring to a specific thing and at other times we mean to denote a possible thing from some restricted class of things. If we are attempting to communicate a method or procedure specification it is essential that the recipient of our communication first understand our conventions for specifying things and second have access to whatever knowledge and skill is required to make use of the specification.

This section deals in part with parameter-passing conventions. Just as I don't have to be explicit when telling a fellow denizen of New Haven to get on a bus whether I mean a particular rundown

smog-belcher or one of a class of such vehicles (e.g. those that travel a particular route), so the planner should have its own internal set of conventions and procedures for unambiguously communicating task specifications. If I have a plan to store an object in a container it should be able to handle a task which specifies a particular container as well as one which simply constrains the choice of container. It doesn't seem reasonable to have two plan specifications; one describing a plan for using a specific container and a second for dealing with an unspecified container only known to meet certain criteria. While this seems simple enough, realizing it is difficult.

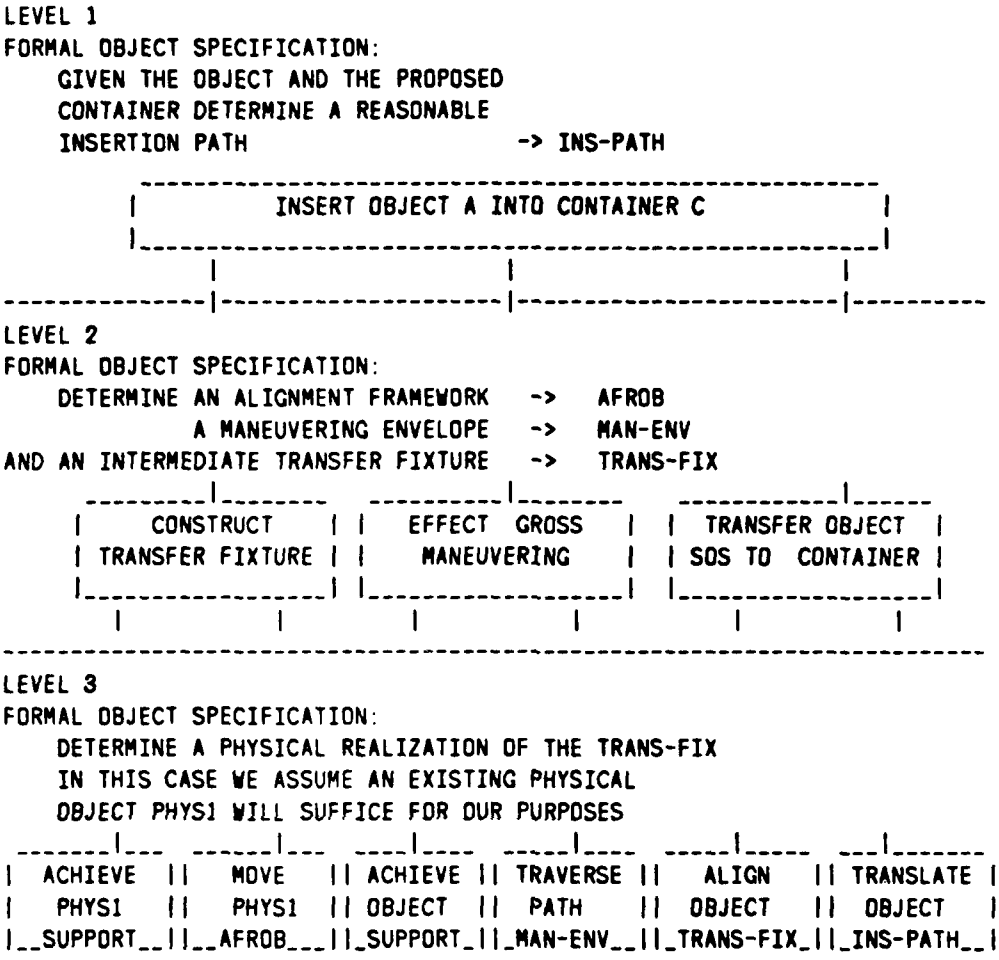
In the following I will distinguish two sorts of items that can appear as arguments in a task specification. First an *object token* or simply an object refers to some actual physical item (e.g. the crate sitting in the corner of the warehouse) or realized concept (e.g. a tolerance for a machined part or a highway speed limit). The second type of argument is called a *formal object* [Sussman 75] [Sacerdoti 77], *uninstantiated parameter* [Sacerdoti 74], or *indefinite-node* [Stefik 79] and serves the function of a variable with certain constraints upon what can be used to fill it (e.g. get on a bus heading downtown). Physical objects possess a special property namely their shape description. A formal object is represented just as an object token, with the exception that constraints imposed upon the instantiation of a formal object are recorded in the data base along with dependency tags indicating which plan imposed the constraint. This is done for a number of reasons:

- it ensures that if a plan is revoked then all of the constraints it imposed will also be removed
- if two plans attempt to impose conflicting constraints on a *formal object* then the plans can be consulted to see if either one or both will relax its constraints.
- if two plans imposing mutually conflicting constraints cannot reconcile their differences an alternate plan(s) can be found to replace one (or both) of the combatants

One problem with the use of formal objects involves how to treat assertions containing them in

the data base. What should the status of such assertions be in the event that the formal objects which they contain are instantiated (i.e. an object satisfying their constraints is specified)? Or conversely how do we interpret them if their formal objects never are instantiated (e.g. I know that he arrived on a train but I have no idea what train?). [Presumably this problem doesn't apply to events corresponding to tasks executed by the planner.] One approach that has been considered is to treat all formal objects just like regular object tokens. If the formal object has a value (i.e. is instantiated to an object token) then the formal object and its intantiated object token are considered identical as long as the instantiation remains justified. If on the other hand the formal object never is identified then its properties are treated as an indication of what we do know (or alternatively as a measure of the planner's ignorance). Implementing this involves modifying DUCK [specifically the equality test in the unification algorithm]. A similar but far more ambitious approach which appears promising is that taken by McAllester in his Reason Utility Package (RUP) [McAllester 82]. RUP computes "substitution simplifications" of terms using equality assertions (e.g. (+ x (- y (+ 1 z))) yields (- y 1) assuming that (= x z)) and supports reasoning about transitive relations.

The use of formal objects relates to a number of issues stressed in the planning literature. The first involves the ability to delay commitment or proceed on a partial specification [Sacerdoti 77] [Stefik 80]. For instance if I have to drive a rented truck from the east to the west coast I may start out with no particular route in mind; only the constraint to stick to interstate highways. I may wait to see what sort of trucks are available from the rental agency and how they handle on the road before making any further route decisions. I might later wish to remove this constraint or at least deviate from it at some point in order to take a scenic sidetrip. Delaying commitment is also used in conjunction with a method of merging constraints imposed by distinct plans as a means of coordinating the design, construction and use of a given instrument or tool or the



Note: a fixture refers to an object or assemblage of objects employed for a special purpose. The term originates from a class of mechanical devices built by tool and die makers for positioning work in machine tools. In the example given in the text, a platform level with the truck's tailgate might suffice for a transfer fixture in the plan expansion illustrated above.

Figure 3-2: A possible plan for inserting an object into a container

allocation and dispersal of some shared resource. Figure 3-2 describes one possible plan for inserting an object into a container in the forklift domain. Assume the task is to load a long I-beam onto an enclosed tractor/trailer. The length of the beam and restrictions on the motion of the forklift prevent us from performing a direct insertion. Instead a fixture (see the note in figure 3-2) is positioned at the rear of the trailer, the I-beam is placed upon the fixture, aligned with the length of the trailer and centered with the rear doors. Finally the forklift inserts the

beam by sliding it along the fixture's surface until it is enclosed within the trailer. The task to construct the transfer fixture may be elaborated before the tasks which will use the fixture. Somewhere in the construction task the formal object corresponding to the fixture will be instantiated but every task which will use it should also have some say in its specification. The fixture may be the right height and position but be shaped so as to prohibit a particular forklift from aligning the beam on the fixture surface. During plan expansion any plan can impose a constraint upon the fixture specification. If two plans impose conflicting constraints their differences must somehow be resolved or alternative plans found.

The use of formal objects can also be tied to the underlying paradigm that the planner relies upon: that of hierarchical planning. Hierarchical planning involves decomposing a domain into a hierarchy of descriptive levels, each level abstracting out some aspects of the domain which can profitably be analyzed in isolation [Sacerdoti 74]. In the context of these various levels we build a task network and a set of plans and choice rules. It is assumed that a set of weak (in that they carry no guarantee of success) but general (in that they potentially serve a wide range of situations) methods will suffice for good performance. In order for such an approach to work plans must be able to place certain constraints upon their arguments and tasks must be able to further restrict the details of plan execution so as to tailor a general plan to a specific situation. As an example suppose as wardens of central park we are given the task of containing a five ton bull alligator escaped from the Lacoste display at Bloomingdales. Our generic containment plan for the task *(contain ?object ?container)* only specifies that the *?container* be of sufficient size to fit the *?object*. However we would like to stipulate that in addition the *?container* be of sufficient strength to confine a big mad bull alligator long enough to get it out of the park (and out of our jurisdiction). The latter constraint will force whatever task is organized under the generic containment plan to choose or design the *?container* to do so in keeping with the present

circumstances.

Formal objects allow planning to proceed even though certain items are not immediately known. Another problem arises when it is necessary to specify a planning assumption referring to a class of possible event or fact tokens. In the plan specification language this occurs in the use of *in-lieu-of* clauses in plan choice criteria. It was rather simple to set up the dependencies for an *in-lieu-of* clause which referred to a completely specified fact type (i.e. an assertion containing no variables or formal objects). If one was to say:

```
(use shop-at-SUP-R-SAV
     in-lieu-of ((currently-spans (open Bimbos-Food-Shop))))
```

Then in order to detect when this assumption is violated it is only necessary to be aware of the duration of fact tokens in which it is believed that *(open Bimbos-Food-Shop)*. However, quite often a plan is dependent upon a more loosely specified class of constraints. For instance I might say that assuming none of the nearby restaurants are open I'll make do with what's in the refrigerator. Or in the forklift domain: in lieu of a physical object between one and two meters in height which will support the load I am proposing to carry and which is located within a two-meter radius of my destination build a composite object meeting these specifications out of available materials. The method whereby the planner manages such constraints is not important here. Suffice it to say that initially all tokens in the kernel satisfying the specification schema are noted and dependency relations are invoked to ensure that their temporal intervals do not overlap the interval associated with the task for which the plan was proposed; in addition a forward chaining rule is set up for the duration of the plan's activation which detects the addition of further assertions matching the constraint specification and updates the dependency relations to suit. The reason for going into all of this is hopefully to demonstrate the utility of being able to use such constraint specifications in planning. The examples posed thus far should provide some justification and hopefully the reader can supply more.

One further application which I think is worth mentioning has to do with profiting from fortuitous occurences [Dehn 84]. Suppose that you have something you want to ask of a particular person but that person is not immediately at hand. You could seek the person out or you could temporarily shelve the task until a better opportunity presents itself. Seeing as you have no idea when such an opportunity might arise, it is not possible to simply reschedule the task. Instead one can settle on a plan of waiting which is based on the assumption that the person in whom you are interested is not in the vicinity. A simplistic plan choice specification is shown in figure 3-3.

```
(to-do (obnoxious-question ?question ?person-to-be-questioned)
   (use wait-for-opportunity
        in-lieu-of ((currently-meets
                     (proximity ?person-to-be-questioned))))
   (else-use blurt-it-out))
```

Or instead we can take advantage of a more direct procedural method invoked by the specification syntax which causes the *blurt-it-out* plan to be expanded immediately upon detecting an adjacent token satisfying *(proximity ?person-to-be-questioned)*

```
(to-do (obnoxious-question ?question ?person-to-be-questioned)
   (use wait-for-opportunity
        in-lieu-of ((fortuitously
                     (meets (proximity ?person-to-be-questioned))
                     blurt-it-out)))
   (else-use blurt-it-out))
```

Figure 3-3: Plan specification for asking obnoxious questions

This assumes that upon hearing the voice or seeing the face of the person you wish to question that you note that you are in the *proximity* of that person. In a more realistic specification the *proximity* predicate test would be replaced by some sensory stimuli which was likely to be triggered on the occasion of seeing or hearing the person of interest. This points up a limitation of the use of explicit planning assumptions. The problem is that often one doesn't know all of the sorts of things which could happen to hamper or facilitate a given plan, or if one could come up

with a list of all such things it would be too lengthy to bother with. The next section deals with a class of tasks whose purpose it is to detect certain conditions occurring within specific temporal intervals.

3.2 Monitor Tasks

Suppose you are robbing a bank for the first time. During the robbery it would be well to be alert for signs of danger. You're not sure just what constitutes a sign of danger but you're pretty sure you'll recognize one when it occurs. In this example the reason for detecting signs of danger would be to initiate some contingency plan (e.g. throw up your hands and surrender or yell "fire") which has been prepared in advance. But the notion of a *monitor* task is quite general [McDermott 77] and can be employed in any context in which it is useful to detect a given condition and perform a particular action on the basis of detecting such a condition. Monitors are especially useful in the context of plan execution monitoring. For instance you might have a task whose purpose it is to detect the occurrence of oil spills and in the event a spill is discovered spawn a task to clean it up. Another typical use of monitors involves execution time creation of further monitors whose task it is to protect a fact or set of facts which just became IN. As an example suppose that you just finished troweling smooth a slab of wet concrete and you wish to protect the fact that the slab is unblemished at least for the period required for the concrete to set hard. Then you might set up a monitor whose purpose it is to detect conditions which might lead to marks in the wet concrete (e.g. people wishing to immortalize their names or amorous exploits, or dogs blindly pursuing their prey).

The current planner has a rather simple syntax for monitor tasks. A monitor consists of:

- a condition specification which includes a fact type (or schema) preceded by a temporal constraint keyword such as *spans* or *meets*
- either a scheduling suggestion, a token status change (e.g. change the status of a DORMANT task to EXECUTE or EXPAND), or another task to be spawned when the condition is met

```
(task-monitor
   condition (meets (location ?widget-delivery loading-bay))
   response  (change-status IMMINENT))
```

Figure 3-4: Task monitor for a rush job

As an example suppose that the forklift is expecting a shipment of widgets needed for a production order which an irate customer is impatiently waiting for. The planner has the task of unloading the widgets immediately upon their arrival but it also has other duties, so it schedules a number of tasks to periodically check the loading bay to see if the truck carrying the widgets has arrived. To accomplish this, the plan for unloading the truck initiates the monitor task shown in figure 3-4 when it is expanded and then sets the status of the token associated with the unloading task to DORMANT pending further notification. Monitor task initialization involves two parts. First a dependency relation is set up just as with planning assumptions so that when the condition is met the action specified in the monitors second field is performed. Next an information gathering task (whose purpose is to attempt to determine if the fact is true) is spawned and scheduled to occur before the task which initiated the monitor. Now the forklift can go about its chores, occassionally executing a plan to see if the truck has arrived. The task spawned by the monitor is similar to other tasks in that different plans may be used to check the bay depending upon the circumstances (e.g. if the forklift is in the proximity of the bay it might just trundle over and look. If on the other hand its working in a part of the plant distant from the bay it might find an intercom and see if anyone answers on the loading dock).

The one difference between a task spawned by a monitor task and other tasks is that monitor spawned tasks are self replicating and self extinguishing. A task spawned by a monitor maintains a record of the event in which the monitor responsible for its existence was initiated. When a monitor spawned task is chosen for execution if this recorded event is past the monitor is simply deactivated. If the recorded event is still to come then the task is executed and a new token is

activated in the kernel with the same task as before. Only the first task spawned in connection with a given monitor is forced to occur before the event which it presumably serves. The frequency and periodicity of the execution of monitor spawned information-gathering tasks is dependent upon the way in which tasks are chosen. Some sort of priority system may be necessary in certain circumstances but the current planner makes use of ad hoc rules, awaiting a better approach.

The primary purpose of tasks spawned in the service of a given monitor task is the extraction of information not explicitly represented in the data base. This process of extraction characteristically involves an attempt to derive the fact listed in the condition specification and add the resulting fact token to the kernel. This usually means (1) situating the robot in a position where it can use the information gathering devices linking the planner to its environment followed by (2) the interpretation of information recieved from these sources.

In the next chapter the importance of information gathering tasks will be stressed along with a discussion of their role in planning and the formulation of a pragmatic model of the planner's environment.

3.3 Aside

This is in some respects orthogonal to the principal issues this paper has set out to address. Nonetheless I include it as an illustration of the potential dangers of mixing levels of abstraction. It has to do with problem solvers versus theorem provers and the dubious utility of reading too much into quantified formulae at the level of plan specifications. Consider the task to clear all the blocks off the table. One might be tempted to represent a plan for solving this problem as:

```
(forall (x y)
    (if (and (block x) (on x table))
        (and (find y)
             (not (= y table))
```

```
(put x y))))
```

In the planner discussed in this paper a plan could be formulated for this task by using a type of iterative plan called a PLOOP. The PLOOP has a set of tasks which are executed followed by a test to determine if some termination criteria is reached. If the criteria is not met the PLOOP regenerates itself and the cycle repeats. The details are unimportant but the specification would be more complicated than the above. Now consider the task to rotate every block on the table 90 degrees. Again the plan might be "solved" quite easily by:

```
(forall (x)
     (if (and (block x) (on x table))
         (rotate ?x pi/2)))
```

The problem with this solution is that it obscures certain parts of the problem by employing a notation which trivializes important aspects of the task domain. A realistic solution might include designing a tour which would move past each block on the table in turn or combining a search task with some means of marking the blocks which have already been rotated.

It is difficult to keep separate the different levels of abstraction employed in implementing a theory of causal reasoning or planning. It is quite easy to be seduced by the seeming power of the theorem proving mechanism and end up using it as a catchall for all that proves difficult (the logic hacker's homunculus). In this paper I have tried to be realistic about the organizational power to be found in time maps and the use of explicit planning assumptions. It is hard for me not to be optimistic about the role such methods might play in building autonomous robot control programs. But thus far only a very small part of a complete theory has been presented. What sort of sensory data could support the sort of plan failure detection mechanism sketched here? How could generalizations be made and even simple forms of learning such as operant conditioning occur? These questions and more must be asked of even a simple proposal such as this. Hopefully in coming years the final judge of planning programs will be observation of the

outward behavior of the mechanisms they control rather than the interpretation of the output of LISP programs which despite our knowledge of the sources of their form and function continue to fascinate and beguile AI hackers. I hope that the reader sees potential in the techniques described in this paper but I also hope that his or her general level of skepticism is raised rather than lowered by the discussions thus far.

The final section sets forth a thesis for a simple adaptive planning scheme which relies upon the TMM approach. This thesis strives to be precise but even so the loose ends far outnumber the concrete proposals. One problem is that we don't really know what we're looking for though a popular claim is that we'll know it when we see it. What it seems is needed is well defined correspondence between the functional components of the behavior we are trying to model and the notations and procedures actually used in building our models. The last chapter attempts to provide such a correspondence and a justification for assuming that the coordinated functional components which the chapter proposes might elicit the sort of behavior expected.

4. Continuous Planning and Execution in Time

4.1 Introduction

This chapter addresses several planning issues: some of which are specific to my view of planning and some of which have wider scope. The two major sections divide these issues into two categories. The first category deals with the continuous elaboration of events in the kernel including the execution of tasks deemed sufficiently elaborated and the addition of new tasks which are activated in the process of elaboration. The first section describes a control structure which supports continuous planning with an emphasis on the garbage collection routines used for removing tokens from the kernel. The second category involves how a planner might go about synchronizing its internal model with events of interest in the real world. With respect to the latter, we first discuss a method of representing and dealing with probable (but likely inaccurate) world models and then turn our attention to a methodology for achieving synchronization (splicing validation tasks into the network) and a cursory look at coherence criteria to be used in driving this methodology.

4.2 Maintaining a moving kernel of active events: the planner's window on the continuum

Its obvious that we never plan out our actions down to the minutest detail. There are levels of description depending on the circumstances which are adequate for detecting those interactions worth modifying ones plans to deal with. In crossing a street, if an assumption that the street is clear is violated by a glimpse of a speeding truck then one should not simply choose to deal with the consequences as they come. On the other hand a detailed simulation of the actual process of locomotion involved in transporting ones body across the intersection is not likely to be required. We're not going to find an algorithm for determining a level of description guaranteed to detect

all interactions from which there can be no recovery. And even if it were feasible to plan every action down to the most detailed level available no model short of an exact description of the world precisely as it will be at execution time could predict success (or failure) with absolute certainty.

The planner described in this paper adopts the same general attitude to planning and execution that McDermott's NASL [McDermott 77] does: choose something from the kernel, expand it and then project its consequences. If the event chosen was a primitive task then expansion corresponds to executing that task. The actual control structure is a bit hairier. It defaults to breadth first expansion until all events are sufficiently detailed (at this point their status would be EXECUTE rather than EXPAND) followed by depth first execution where the kernel is seen by the program as a flat lattice of events ordered along one dimension by level of description and along a second by temporal order. The control structure supports the use of information gathering tasks. That is, all events in the kernel with the exeception of certain information gathering tasks may be DORMANT pending further information. These investigatory tasks are then executed with the side effect of causing certain of the pending tasks to have status EXPAND. The method is similar to that used in the fortuitous planning example of the previous chapter. I might have a task to occupy a strategic position in a game like chess or go which involves two tasks one to assess your opponents strength and the other to move (sereptitiously) into a preliminary configuration. The second task must wait for the first to be executed before it can even be planned.

During the process of elaboration, new events (some of them tasks like the above information gathering tasks) are continually being added to the kernel. However the kernel is expensive to maintain both in terms of storage space for all the *derived-before* assertions and in terms of the time required to update the transitive closure and set up consistency mandates. Luckily once we

have determined when a token has occurred (in the case of a planner: the event token associated with a task has been "executed") it is no longer necessary that it remain in the kernel. The kernel is useful primarily for exploring the ramifications of an event's occurring at one point in time rather than another.

Extricating a token from the kernel requires unraveling just that part of the intricately interwoven dependency network which refers only to that token without disturbing other tokens still active in the kernel which might depend upon that token or its effects. As we said earlier elaboration is the process of expanding (detailing) and projecting (inferring the causal effects of) event tokens based upon assumptions derived from other tokens in the kernel. Performing any modifications to the kernel must be done with precision, in order to avoid unwarranted assumption-violation warnings. The functions responsible for token garbage collection are listed in the appendix and I won't try the reader's patience with a detailed description of their operation. I will, however, give a simple example to demonstrate some of the problems involved.

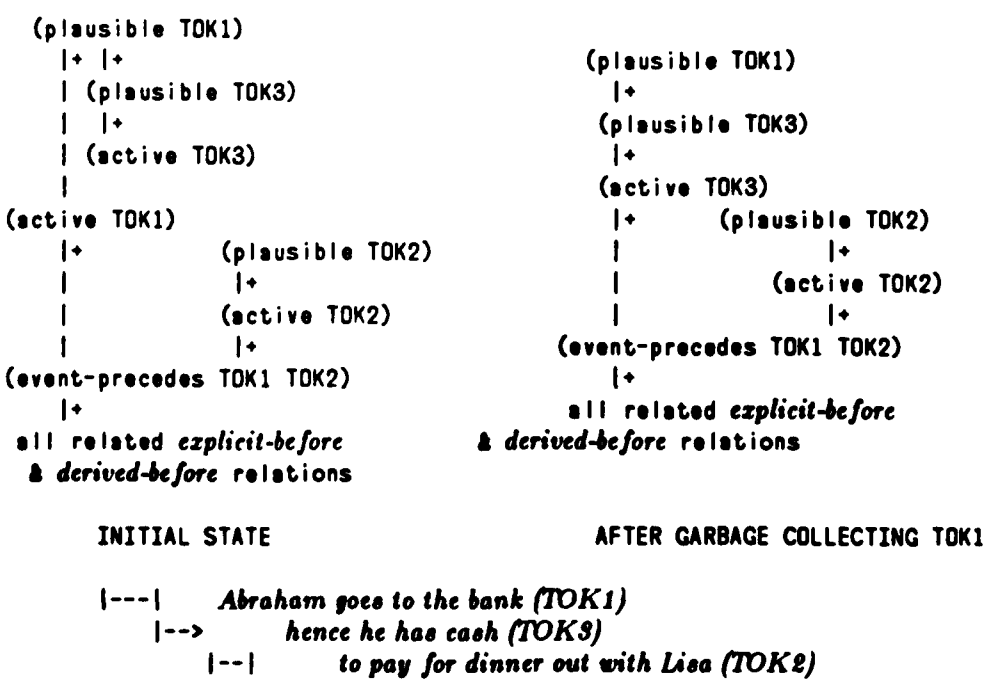


Figure 4-1: Simple example illustrating token garbage collection

Suppose we know that Abraham went to the bank and cashed his paycheck. Let this correspond to the event token TOK1. We also know that he took Lisa out to a very expensive restaurant and this event we associate with TOK2 (see figure 4-1). Based upon our current knowledge of Abraham's weekly spending habits we assume that since it is late in the week TOK1 must have occurred before TOK2. Otherwise we can't imagine how Abraham could have paid the check. Now TOK1 gives rise to a fact token TOK3 asserting that for some time following TOK1 Abraham has a reasonable amount of ready cash. It is upon the *tassertion* associated with TOK3 that we have based our projection that Abraham will be able to pay the maitre d' when the time comes.

Now suppose we wish to remove TOK1 from the kernel: we're still interested in exploring what might have happened at dinner but were not interested in what came before. Two things to keep in mind: (1) the only reason that we know TOK3 begins before TOK2 is that we assumed TOK1 occurred before TOK2 and (2) if at some future time it is determined that Abraham did not get his paycheck cashed we want to be able to retract the assertion *(plausible TOK1)* and have it wipe out all of its effects in particular TOK3 (thus alerting the program of its invalidated assumption with respect to TOK2). The token garbage collection routine handles this by finding all *explicit-before* assertions threatened by the removal of TOK1 and modifying their justifications so that they will continue to be IN contingent upon *(plausible TOK1)* and those tokens still active in the kernel that depend for their temporal position upon these *explicit-before* assertions.

As was said earlier, the decision of what is to be retained in the kernel and what is to be discarded is pretty much left up to the discretion of the programs using the TMM. The garbage collection routine can extract a token from the kernel but it must be told what tokens are superfluous. In the planner described in this paper the decision of what to remove is built into

our model of integrated continuous planning and execution and our representation of possible worlds. The next section outlines this representation and the criteria for token removal.

4.2.1 Planning in Time

Imagine the planning program as traversing a great network of events. From the planner's point of view the network is a tree and at any one time the tree only branches into the future. As soon as it reaches a fork it must decide which branch to take; the choice it makes determines its best guess of what happened, the other branches are only "could've beens" (the program may later be forced to revise its guess - we'll return to this). Each event that the planner is aware of is marked on the tree by two points (the event's beginning and ending) located on the branch in which the event is to occur or to have occurred . The planner cannot always tell whether a given point occurs before another (i.e. the tree imposes a partial order). However the planner can consider any ordering of previously unconstrained points it wishes (e.g. the planner might believe that one point precedes or follows another and choose to see it that way or assuming that an event is under the planner's control, it can decide to schedule that event before or after another event). The TMM kernel is the part of the network of events that the planner can see. Its decision as to what it wishes to see corresponds to a hypothesis about how events are to unfold. The branches constitute different hypotheses being considered. Some decisions constrain others and hence if the planner chooses to see things as happening one way it cannot help but see those things it knows will follow as consequences.

The planner always has the option to either look further up a branch (projection) or to look closer at that portion of the branch currently in view (expansion). At some level expansion consists of actually carrying out certain primitive steps (e.g. accelerating, reversing or lifting the fork in the case of the forklift domain). At the point when the planner for some reason decides to exercise some physical option, it has, in effect, chosen a branch of the tree. I call this

execution-time enforced scheduling. At any one time the planner has before it some number of events which correspond to tasks that have no known events that precede them in the active kernel. In the garbage collection algorithm these are referred to as root-tokens. The planner chooses one of these root-tokens denoting a task and constrains all the others to follow. The imposed schedule can easily be retracted if it is seen to lead to problems. The purpose of imposing it is simply to force the kernel to reflect the fact that the past is a complete (linear) order and detect any interactions which would otherwise go unnoticed. Execution-time enforced scheduling attempts to funnel our predictions of what is be into a linearized representation of what has been.

Let's suppose that the planner decides to execute the root-token task it has chosen. Then, assuming that the associated plan does not for some reason meet with disaster, the root-token will be garbage collected upon completion of all subtasks serving the associated plan. If the plan is determined to be failing irrevocably, then either a new plan will be selected or the task itself will be deemed not worth further effort and be revoked. In this way the planner moves along the time network illuminating only that part which it considers of immediate value: continually splicing in new tasks and disregarding completed ones.

The branches in our time network have a special significance in McDermott's theory of planning [McDermott 82]. They correspond to what he calls *chronsets* and situating itself on a branch which presents a reasonably accurate model of the world is the planner's single most pressing intellectual activity.

4.3 Synchronising an internal representation with real events

A *chronicle* [McDermott 82] can be thought of as a single continuous path through the time network. The planner maintains descriptions of alternative chronicles and since these

descriptions are necessarily incomplete, many chronicles will in general fit a given description. We call the set of chronicles fitting a description the *chronset* of the description. What appear to the planner as a number of distinct individual paths branching before it, constitute, not single paths, but whole classes of possible futures corresponding to chronsets. A *sub-chronset* is partial description subsumed by other more detailed descriptions and their respective chronsets. A sub-chronset was at one time a regular chronset until for some reason it was split (to examine different possibilities stemming from a single source of uncertainty) into two or more chronsets each of which incorporated it as a sub-chronset. The set of all chronsets which are not subsumed by some other chronset is called the set of *terminal chronsets*. The TMM routines manage the events in this set of *terminal chronsets* so as to maintain each event in the most general chronset serving all those chronsets referring to that event. Every event token has a unique elaboration and if two hypotheses concerning the temporal elaboration of one event are to be explored then two tokens are created each with its own projection and expansion. The chronset management routines make the operation of hypothesizing alternate futures reasonably inexpensive. The ability to switch rapidly between chronsets is managed using the data pool mechanism of DUCK [McDermott 81] which is similar in many respects to the CONNIVER context mechanism [McDermott 73]. I won't spend any more time talking about how chronsets are managed but I will briefly discuss the role they might play in detecting and recovering from planner errors. We'll begin by exploring some of the many ways that the planner can be led astray.

The planner is continually making assumptions about events, their interactions with one another and the occurrence and persistence of their effects. Its ability to anticipate or predict actual events relies upon it having the right information at hand. Knowing that X causes Y in the context of Z is not sufficient to predict Y. The planner must "know" that X has occurred in an interval in which Z is believed to be true in order to infer Y. In order to "know" anything the

planner must actively seek out (search for) observable phenomena from which to conclude the necessary facts. This implies that either (1) the planner has already predicted that something will occur and is only seeking to confirm its conjecture or (2) it is sufficiently important to its purposes that something does (or does not) occur and it has deemed it a reasonable expenditure of energy to interrupt its other activities to search for certain evidence. There is a certain sense in which it can be said that we know nothing with certainty (except analytic truths). Certain modern philosophers have been able to counter the arguments of the skeptic claiming that we can know nothing by appealing to a foundation of basic assumptions about the world and our teleologically biased view of it. I invite the reader to read one of the philosophical references cited in the bibliography if he is interested in further pursuing this [Rescher 77] [James 07]. For the process of acquiring and assimilating information to converge on a realistic or pragmatic view of the world the planner must assume a great deal about the world and its own ability to observe the effects of events occurring in the world. The planner relies upon:

- Explicit assumptions concerning the occurrence of events and the persistence of facts in the context of the modeled world.
- Implicit assumptions about facts and events in the "real" world bound up in the planner's control structure and the store it places in its ability to perceive the world about it.

If the planner fails (we will assume that in order to fail it must realize that it has been unsuccessful in achieving one or more of its tasks) it does so for one of two reasons:

1. failure to anticipate a possible interaction which was understood but simply not deemed worth probing for the signs of its manifestation. This implies the plan selection criteria is at fault (e.g. I assumed that flicking the switch on the appliance was sufficient for its operation but I neglected to see that the cord was plugged in).
2. an inappropriate assumption or belief concerning the state of the world based on faulty sense data or incorrect inference from extracted features. The erroneous assertion may come to light as the program attempts to validate its assumptions in searching for the cause of its failure (e.g. I believed that if the lights were on in the house someone would be home but I find that there is at least one exception to the rule). In the worst case something unexpected might occur which the program may or may not have knowledge of but which brings about an interaction it has no way of explaining (e.g. a jet flies low over the farmhouse and later I find the china decimated but I fail to make the connection).

Each of these types of errors will be evident as a failure to choose a suitable chronset from all those possible. Sometimes the planner was aware of a chronset which would have led it to success. At other times it can construct an appropriate chronset after the fact but can only conjecture as to the generality of the model it has constructed and its potential utility in helping it avoid future errors. A successful planner will be able to synchronize itself with a "reasonable" chronset in the midst of time passing and model/world discrepancies noticed. A reasonable chronset for a given planner is one which provides a description of the world sufficient to guide the choice and execution of plans for achieving the planner's set tasks. In order to understand what might be entailed in such a synchronization process let's take a look at how new chronsets are created and how they can be distinguished and in some cases discarded.

First some observations:

- chronsets split at decision points: places where uncertainty exists and hence we conjecture alternate projections of the future. The uncertainty can be due either to lack of knowledge about whether or not an event will occur or how it will turn out. These two cases are related in that the question of whether or not an event E1 will occur will always be relative to some other event E2 and this latter (though likely chronologically prior) event has projections in which E1 occurs and those in which it doesn't.
- there are events (corresponding to tasks) that the planner has scheduled but which may or may not be executed
- there are events which the planner is expecting but may or may not occur
- presumably the scheduler chooses a particular chronset first on the basis of its expected conformance to reality and secondly on the basis of the expected viability of its proposed plans for realizing its goals. It likely rejects those chronsets which fail to meet with its physical (sensory) experience of reality
- let us suppose that the planner has chosen a chronset and executed a plan which in all likelihood will include some information gathering. It will obviously be aware of events which it has itself executed though not necessarily aware of the validity of their projected effects (other events and facts). Some events it may believe will occur but it will not verify that they indeed occurred . Event and fact tokens corresponding to the effects of such events may persist in the database despite their failure to coincide with reality. If later the planner attempts to verify these effects and fails, it may question the projection which gave rise to the causing event (certainly successful verification should be added to the support for an hypothesized event). It might also try to conjecture some unforeseen interfering event. An event conjecture can be excised from a chronset whenever the effects the conjecture proposes are invalidated or in the case of a plan, it is implicated in a conflict and deemed unpatchable (or at

least not worth patching). In any case all events corresponding to tasks organized by some conjecture are excised along with that conjecture.

If the planner is to synchronize its internal model with the real world it must have have some means of distinguishing fact from fantasy. Why should I believe in an event E1 having occurred ? Presumably because I believe that some event E2 which is believed (under certain circumstances) to cause E1 has occurred and that nothing further has occurred which might prevent E1 occurring. I might also believe in an event's past occurrence based upon validation of its effects. For instance I might believe that a train has recently traveled over a section of railroad track based upon sensing a vibration and warmth in the steel rails.

Suppose that I wish to know if anyone recently dove into a swimming pool. I might look to see if anyone is presently in the pool. This however would not be conclusive as the swimmer might have entered the pool by means other than diving. If I believed that a dive must result in a splash then I might check around the deck for puddles of water. Unfortunately a dry deck does not necessarily disconfirm the hypothesis that a person sighted in the pool, entered by diving. The water could have been deposited upon the deck and evaporated before I could observe it.

There are facts which if true signal the coming of an event. There are events which if they occur generally signal that certain other events will follow or co-occur (their respective temporal intervals overlap). Finally there are facts and events which occur as results or consequences of an event. When I assume that an event E will occur it is quite likely that I will make decisions based upon this belief. In particular I may choose a plan relying upon a fact which will become true as a result of E occurring. Now it is completely plausible that I set myself the task of monitoring E to make sure that it unfolds as I have projected. That is, I will check to see that the conditions leading to the event occur and that certain of its consequences are observed. Such event validation plans will likely be tailored to the super task which they subserve and hence be

selective in those effects they expend energy confirming. Validation plans can report confirmation and yet the event fail to occur, as in the case of a swimmer who uses the ladder to enter the pool after a summer rain storm. Validation plans can also disconfirm an event which actually occurred : for example the sun evaporates the water on the pool deck from a diver's splash before we can observe it.

The idea of scheduling tasks whose purpose it is to validate and detect the occurrence of events and their effects is a crucial part of planning. It would be nice if our senses and ability to discriminate were so reliable and rapid that we could use the simple criteria in figure 4-2 for predicting events and determining their past occurrence.

Unfortunately it is hardly ever practical to check every condition (as in the first rule in figure 4-2) and even if it were possible it would be necessary to check infinitely often in the interval preceding the event of interest to make absolutely sure that the conditions were in place. The *likely-occurred* predicate assumes that all effects are detectable and that furthermore they will persist at least until we have time to validate them. The problems of validation and choosing a reasonable model is not apt to be solved by so simple an approach. I expect that time maps can play a significant role in a pragmatic theory of causality: a first step toward solving these problems. I am currently exploring realistic projects in the fork lift task domain which depend upon a partial solution to some of the issues raised above in order for a program to perform well over time. The last section sketches a proposal for an adaptive planner operating in some task domain and serving as a test bed for validation strategies and pragmatic causal theories.

```
(forall (event1 interval1)
  (if (exists (event2 interval2)
         (and  (meets interval2 interval1)
               (likely-to-occur event2 interval2)
               (ecause event2 event1)
               (forall (p)
                  (if (condition-for-causing p event2 event1)
                      (exists (interval2)
                           (and  (overlaps interval2 interval1)
                                 (t interval2 p)))))))
      (likely-to-occur event1 interval1)))
```

Paraphrase: You can conclude that event1 is likely to occur in interval1 if you know of an event (event2) likely to occur in an interval (interval2) immediately preceding interval1 such that event2 is known to cause event1 under certain conditions currently known to hold.

```
(forall (event1 interval1)
  (if (forall (p)
         (if (effect-of p event1)
             (exists (interval2)
                 (and  (overlaps interval1 interval2)
                       (t interval2 p)))))
      (likely-occurred  event1 interval1)))
```

Paraphrase: You can conclude that event1 has likely occurred in interval1 if you know that its effects are true in intervals overlapping interval1.

Figure 4-2: Possible rules for establishing facts about events

5. Conclusions

This paper is an amalgam of conceptual approaches, programming techniques and hints at where my primary research is heading. The whole is presumably tied together by the notion of time map maintenance and its potential for application in planning programs. In the following I will try to sketch a more ambitious research proposal employing time maps.

When the planner was in an early stage of development I was asked to describe how the planner would handle a simple blocks world problem. My answer, such as it was, relied upon a naive conception of monitor imbued with immense power and quite as convenient as the homunculus for avoiding real issues. In retrospect I believe that I had at that time an aversion to "mechanism". Whenever it became apparent that here was an area in which humans performed with extraordinary range and flexibility then it seemed obvious that no simple mechanism would suffice to explain it and certainly no measly five thousand lines of code would serve to model it. The problem as I now see it is that I sought to build a model of adaptive environmental response aiming at the wrong level of functionality. I wanted to construct a mechanism which elicited that sort of behavior. However it was obvious that I could not adequately describe the functionality of such a mechanism except by reference to the behaviors themselves. Once I ceased looking at the behaviors themselves and began looking at why they might have been evoked given the environmental stimuli, another sort of functionality became apparent. Some of the component parts even appeared ripe for mechanization. What were the *relevant* stimuli of an experience and how were they distinguished from the welter of other stimuli available? How can *relevance* be defined in terms of teleology and pragmatics? What sort of impression must the stimuli have made upon the organism and what part of the initial stimuli must be retained and how is it to be encoded in order that the organism recall an appropriate pattern of response under circumstances in which executing that pattern will be to the organism's advantage?

Assuming such mechanisms could be constructed the problem becomes one of coordinating the behavior of these component processes by means of conventions for passing information between basic, functionally isolated, processes and supplying a set of rules and biases to form an initial praxis. A similar change in perspective served to change the emphasis of neurophysiology earlier in this century.

In the exitement following Broca's and Wernicke's discoveries supporting the centralization of speech and language comprehension in the left hemisphere an attempt was made to map out the entire surface of the cerebral cortex to localize the functional centers of human thought. Centers of "ideation", mood and volition were "discovered" and their boundaries carefully drawn. However no satisfactory mechanism was developed to explain this organization of functions and psychologists and physiologists employed such catchall terms as *association* (of sensations and ideas) and *spreading activation* to describe what they couldn't explain. The early work of Pavlov excited few psychologists interested in complex cognitive skills and those it did were all too frequently prone to explain everything in terms of those few simple mechanisms that Pavlov had demonstrated. But other mechanisms were discovered: lateral inhibition, spatial summation and the complex neural feedback circuitry in the cerebellar cortex to name but a few. These mechanisms enabled researchers to begin exploring a different level of functionality; a level that sought to explain how the organism interacted with its environment in such a way as to ensure its continued evolutionary success. The result was a new view of functionality. A. R. Luria writes in the opening chapter of "The Higher Cortical Functions in Man" [Luria 66]:

> **"Since Pavlov advanced his reflex theories, the word "function" has come to mean the product of complex reflex activity comprising: uniting exited and inhibited areas of the nervous system into a working mosaic, analyzing and integrating stimuli reaching the organism, forming a system of temporary connections, and thereby ensuring the equilibrium of the organism with its environment. This is why the concept of localization of function has also undergone a radical change and has come to mean a network of complex dynamic structures or combination centers, consisting of mosaics of distant points of the nervous system, united in a common task."**

The time map mechanism is a simple functional unit. It serves the overall planning program by maintaining a coherent account of the effect of actions and events hypothesized to have occurred . The time map mechanism directly influences the functional unit dealing with event projection by keeping explicit track of the relative positions of tokens representing facts true over time and events occurring in time. In the present planner the process of event projection and plan formulation is highly automated. At this level a response to a complex of stimuli recorded in the kernel structures is formulated according to whatever rules the system possesses. In the process of event projection whatever messages or suggestions are found tagged to data structures invoked by inference rules are passed up to higher levels of control. Figure 5-1 provides a functional schematic of the control organization being proposed.

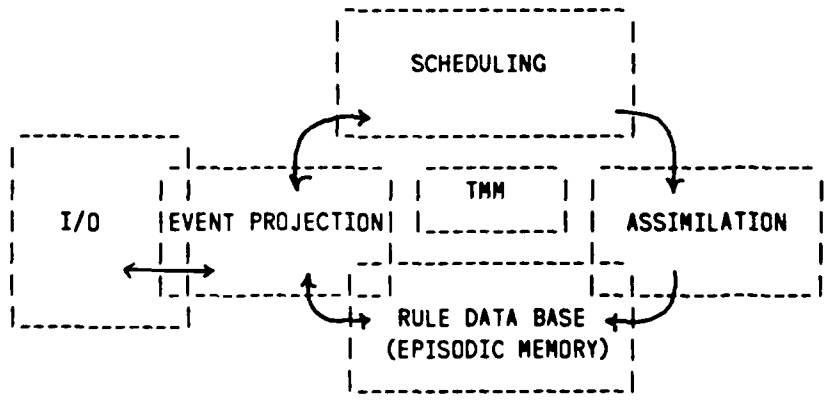


Figure 5-1: A Schematic Illustrating the Organization of Basic Functional Units in an Adaptive Planner

The next higher level of control is presently the least well understood but I can provide a reasonable sketch. The function most clearly linked to this level is scheduling (for a discussion of the role of scheduling in planning see [Miller 83]). The projection mechanism is responsible for recalling past problems and spawning new tasks to gather information and deal with expected contingencies. The scheduling unit deals with event complexes which are either relatively novel

or simply not fully integrated into the event projection process. In addition the scheduler, having sought to validate the predictions of the projection process, tags those experiences which can be compiled into causal knowledge and suggests organizational changes to the data structures which feed the projection process. The idea is that the scheduler only introduces further constraint into the kernel tokens or modifies the sequence of tasks when it is testing a causal hypothesis law and the form of the test is always dictated by the success of the planner's current tasks. The only time it tests a causal hypothsis is when it is sent a message from the projection process which was attached to an event being adding to the time map. Such messages are attached to an event when a validation task has reported that an event which was believed to have occurred appears not to have and tags the record of this episode with a message to the scheduler to be read the next time this event is suspected to occur.

Is the scheduler just a repository for all poorly understood functions: a new home for the homunculus? I think not. There is an obvious cycle of processing. The cycle is nurtured by the regularity and predictability of the data which the planner is designed to feed upon. All the system is really doing is learning to mimic its environment in order to anticipate the conditions for achieving success. The object is for what actually does happen to coincide with the planner's internal model of what it believes will happen coordinated with whatever role it proposes to play in order to edge things in its favor. Success is manifest as a lowering in traffic around the cycle. Causality is just the proclivity of one thing to happen in the context of others in a demonstrably regular or predictable fashion.

The idea of actively instigating validation tasks is central to this approach. Most planners passively accept whatever data is presented to them, assuming that it will be adequate for their processing needs and that irrelevant data can easily be recognized and ignored. But the content and organization of the incoming data need not bear any relation to the immediate informational

needs of the program nor to the way in which the program internally organizes its experience. The idea behind validation tasks is that the planner continually attempts to corroborate its expectations so as to synchronize its internal model with the external environment wherever it is to its advantage to do so (i.e. whenever coordination between the model and the modeled world will help to achieve the planner's tasks). This sort of active information gathering is critical to planning in complex task domains.

It is my aim to demonstrate the utility of the approaches outlined in this paper in a working program within the next year and a half. In order to accomplish this I will have to address some of the many problems I have avoided in this paper: the need for metric constraints or a method of comparing the relative duration of tokens and representing continuous processes to mention some of the most obvious omissions. By making such a demonstration I hope to answer a few questions which were only raised in this paper as well as build a platform from which to pusue these problems further. I appreciate the reader's perseverence in making it this far and I welcome any comments or criticism.

I. Appendix: Time Map Garbage Collection

The following algorithm is used for removing tokens from the kernel without disturbing the overall dependency structure of the events remaining in the kernel. The basic operation was described in the chapter on planning. The algorithm is presented in LISP format with function names chosen for clarity. No particular dialect of LISP is intended though the data structure syntax is that of NISP [McDermott 83]. The form *(: (<structure type> <structure slot>) <structure>)* is used for accessing and setting slots (fields) in a structure (record). For instance *(: (TOKEN begin) tok)* refers to the beginning of the interval associated with *tok*.

A number of predicates require some explanation. *(During tok1 tok2)* asserts that *?tok1* occurs during *tok2*. The schema *(during ?dtok1 tok)* is used for fetching all sub-events in the expansion of *tok*. The form *(token-interval ?tok ?begin ?end)* is used for finding the beginning and ending points of a token given the token (or the token given its beginning and ending points). *(Thnot P)* succeeds only if *P* is neither in the data base nor derivable from rules and assertions in the data base. *(Precedes tok1 tok2)* refers to the fact that *tok1* temporally precedes *tok2*. The algorithm exactly as presented will not work in DUCK as I have omitted quotes and backquotes which would confuse readers unfamiliar with our local syntax. Just assume that the *fetch* function magically knows what to evaluate and what not to. Finally in an expression of the form *(:= foo (baz *-*))* the *-* refers to *foo* in order that *foo* be evaluated only once.

```
(define  gc-past-tokens-tm (tok)
 (let ((begin (: (TOKEN begin) tok)) (end (: (TOKEN end) tok)))
  ;; make sure it is past: all its sub-events should have dissappeared.
     (if
      (fetch (thnot (and (during ?dtok tok) (active ?dtok))))
  ;; if so then update the set of the current chronset's root tokens.
      (do
        (for-each-ans (fetch (and (token-interval ?tok ?pbegin ?pend)
                                  (before ?pend end)))
              (gc-past-tokens-tm ?tok))
        (for-each-ans (fetch (and (precedes tok ?tok1)
                                  (thnot (precedes ?tok2 ?tok1))))
```

```
            (push ?tok1 (: (CHRONSET root-tokens) *current-chronset*)))
        (wean-dependent-tokens-tm tok)
        (:= (: (CHRONSET subtree-roots) *current-chronset*)
            (remove tok *-*))
        (:= (: (CHRONSET tokens) *current-chronset*) (remove  tok *-*))
        (erase (active tok))
        (for-first-ans (fetch (during tok ?tok1))
            (gc-past-tokens-tm ?tok1))))))

(define wean-dependent-tokens-tm  (tok)
 (let ((begin (: (TOKEN begin) tok)) (end (: (TOKEN end) tok)))
  (ans-support (+ (plausible tok))
   ;; CASE I     |-E2-|  |-E3-|
   ;;           |...|--E1-->
   ;; where (EXPLICIT-BEFORE E1 E2)
   ;;       (EXPLICIT-BEFORE E2 E3) and E2 is to be garbage collected.
   (for (gc-pt in (list begin end))
    (do (let ((dependents ()) ((inheritance ())))
           (for-each-ans (fetch (explicit-before ?past-pt gc-pt))
             (push  ?past-pt dependents))
           (for-each-ans (fetch (explicit-before gc-pt ?future-pt))
             (push ?future-pt inheritance))
           (for (past-pt in dependents)
            (do
             (for (future-pt in inheritance)
               (do
                (for-each-ans (fetch
                    (and (token-interval ?tok1 past-pt ?pend)
                         (or (token-interval ?tok2 future-pt ?fend)
                             (token-interval ?tok2 ?begin future-pt))))
                  (ans-support (+ (active ?tok1) (active ?tok))
                   (premiss (explicit-before past-pt future-pt))
                   (premiss (before past-pt future-pt))))))))))
   ;; CASE II    |-E2-|  |-E3-|
   ;;                 |--E1->
   ;; where (EXPLICIT-BEFORE E2 E3)
   ;;       (MEETS E2 E1) and E2 is to be garbage collected.
           (if (eq gc-pt end)
               (for (future-pt in inheritance)
                (for-each-ans (fetch
                   (and (token-interval ?mtok end ?mend)
                        (or (token-interval ?tok2 future-pt ?fend)
                            (token-interval ?tok2 ?begin future-pt))))
                   (ans-support (+ (active ?mtok) (active ?tok2))
                         (premiss (explicit-before ,end ,future-pt))
                         (premiss (before end future-pt))))))))))))
```

Remove the garbage collected token from the kernel associated with the chronset currently being

referenced. This is also used for removing tokens which are deactivated for reasons other than garbage collection; for instance if a plan is scrapped all of the tokens associated with the tasks that the plan organized must be removed from the kernel of the chronset in which the plan was proposed.

```
(rule  deactivate-garbage-collected-token
       (if-erased  (active ?tok)
                   (:= (: (CHRONSET tokens) *current-chronset*)
                       (remove ?tok *-*))))
```

References

[Allen 80] Allen, J.F. and Perrault, C.R.
Analyzing intention in utterances.
Artificial Intelligence 15(3), 1980.

[Allen 81a] Allen, James.
Maintaining knowledge about temporal intervals.
Technical Report TR86, U. of Rochester Department of Computer Science, 1981.

[Allen 81b] Allen, James.
A general model of action and time.
Technical Report TR97, U. of Rochester Department of Computer Science, 1981.

[Charniak 81] Charniak, Eugene.
A common representation for problem-solving and language-comprehension information.
Artificial Intelligence 16(3):225-255, 1981.

[Dehn 84] Dehn, Natalie.
Creative Reasoning and Invention in a Reconstructive and Dynamic Memory. 1984.
Forthcoming PhD Thesis.

[deKleer 77] de Kleer, Johan, Doyle, Jon, Steele, Guy L., and Sussman, Gerald J.
Explicit control of reasoning.
Memo 427, MIT AI Laboratory, 1977.
Also in *Proc. Conf. on AI and Prog. Lang.*, Rochester, which appeared as *SIGART Newsletter* no. 64, pp. 116-125.

[Doyle 79a] Doyle, Jon.
A truth maintenance system.
Memo 521, MIT AI Laboratory, 1979.

[Doyle 79b] Doyle, Jon.
A truth maintenance system.
Artificial Intelligence 12(3):231-272, 1979.

[Fikes 71] Fikes, Richard and Nilsson, Nils J.
STRIPS: A new approach to the application of theorem proving to problem solving.
Artificial Intelligence 2:189-208, 1971.

[Hendrix 73] Hendrix, Gary.
Modeling simultaneous actions and continuous processes.
Artificial Intelligence 4:145-180, 1973.

[Hewitt 71] Hewitt, Carl.
Procedural embedding of knowledge in PLANNER.
In *Proc. IJCAI 2.* IJCAI, 1971.

[James 07] James, William.
Pragmatism.
New York, 1907.

[Luria 66] Luria, A. R.
Higher Cortical Functions in Man.
Basic Books, 1966.

[McAllester 82] McAllester, David A.
Reasoning Utility Package User's Manual.
Technical Report 667, MIT AI Laboratory, 1982.

[McDermott 73] McDermott, Drew V. and Sussman, Gerald J.
The Conniver Reference Manual.
TR 259a, MIT AI Laboratory, 1973.

[McDermott 77] McDermott, Drew V.
Flexibility and efficiency in a computer program for designing circuits.
Technical Report 402, MIT AI Laboratory, 1977.

[McDermott 78] McDermott, Drew V.
Planning and acting.
Cognitive Science 2(2):71-109, 1978.

[McDermott 81] McDermott, Drew V.
Contexts and data dependencies: a synthesis.
1981.
To appear in IEEE Transactions on Pattern Analysis and Machine Intelligence.

[McDermott 82] McDermott, Drew V.
A temporal logic for reasoning about processes and plans.
Cognitive Science 6:101-155, 1982.

[McDermott 83] McDermott, Drew V.
The NISP Manual.
Technical Report 274, Yale University Computer Science Department, 1983.

[Miller 83] Miller, David.
Scheduling Heuristics for Problem Solvers.
Technical Report 264, Yale University Computer Science Department, 1983.

[Rescher 77] Rescher, Nicholas.
Methodological Pragmatism.
New York University Press, 1977.

[Rieger 76] Rieger, Charles.
An organization of knowledge for problem solving and language comprehension.
Artificial Intelligence 7, 1976.

[Sacerdoti 74] Sacerdoti, Earl.
Planning in a Hierarchy of Abstraction Spaces.
Artificial Intelligence 7(5):231-272, 1974.

[Sacerdoti 77] Sacerdoti, Earl.
A Structure for Plans and Behavior.
American Elsevier Publishing Company, Inc., 1977.

[Stefik 79] Stefik, Mark J.
An examination of a frame-structured representation system.
In *Proc. IJCAI 6*, pages 845-852. IJCAI, 1979.

[Stefik 80] Stefik, Mark J.
Planning with Constraints.
Technical Report STAN-CS-80-784, Stanford Computer Science Department, 1980.

[Sussman 75] Sussman, Gerald J.
Elsevier Computer Science Library: A Computer Model of Skill Acquisition.
American Elsevier Publishing Company, Inc., 1975.

[Tate 75] Tate, Austin.
Using Goal Structure to Direct Search in a Problem Solver.
Technical Report TR86, U. of Edinburgh Machine Intelligence Research Unit, 1975.
PhD thesis.

[Tate 77] Tate, Austin.
Generating Project Networks.
In *Proc. IJCAI 5.* IJCAI, 1977.

[Wilensky 83] Wilensky, R.
Planning and Understanding.
Addison-Wesley, Reading, Mass, 1983.


DAT
ILM